Claude Delannoy

Plus de 50 000 exemplaires vendus

# Programmer en LANGAGE C

## Cours et exercices corrigés

5e édition

EYROLLES

# Programmer en LANGAGE C

5e édition

Cet ouvrage est destiné aux étudiants débutants en langage C, mais ayant déjà quelques notions de programmation acquises par la pratique, même sommaire, d'un autre langage.

Les notions fondamentales (types de données, opérateurs, instructions de contrôle, fonctions, tableaux…) sont exposées avec un grand soin pédagogique, le lecteur étant conduit progressivement vers la maîtrise de concepts plus avancés comme les pointeurs ou la gestion dynamique de la mémoire.

Chaque notion importante est illustrée d'exemples de programmes complets, accompagnés de résultats d'exécution. De nombreux exercices, dont la solution est fournie en fin d'ouvrage, vous permettront de tester vos connaissances fraîchement acquises et de les approfondir.

Cette cinquième édition inclut les nouveautés des dernières versions de la norme ISO du langage (C99 et C11).

## À qui s'adresse ce livre ?

- Aux étudiants de DUT, de BTS, de licence ou d'écoles d'ingénieur.
- Aux autodidactes ou professionnels de tous horizons souhaitant s'initier à la programmation en C.
- Aux enseignants et formateurs à la recherche d'une méthode pédagogique et d'un support de cours structuré pour enseigner le C à des débutants.

Ingénieur informaticien au CNRS, **Claude Delannoy** possède une grande pratique de la formation continue et de l'enseignement supérieur. Réputés pour la qualité de leur démarche pédagogique, ses ouvrages sur les langages et la programmation totalisent plus de 500 000 exemplaires vendus.

## Sommaire

Introduction au langage C • Les types de base du C • Les opérateurs et les expressions • Les entrées-sorties : *printf*, *scanf* • Les instructions de contrôle : *if*, *switch*, *do…while*, *while*, *for…* • La programmation modulaire et les fonctions • Variables locales et variables globales • Les tableaux et les pointeurs • Les chaînes de caractères • Les structures et les énumérations • Les fichiers • Gestion dynamique de la mémoire : fonctions *malloc*, *free*, *calloc*, *realloc* • Le préprocesseur • Les possibilités du langage proches de la machine : opérateurs de manipulation de bits, champs de bits, unions • Les principales fonctions de la bibliothèque standard (*stdio.h*, *ctype.h*, *string.h*, *math.h*, *stdlib.h*) • Corrigé des exercices.

18 €
**www.editions-eyrolles.com**
**Groupe Eyrolles** | Diffusion Geodif

# Programmer en langage C

5e édition

## Cours et exercices corrigés

AUX EDITIONS EYROLLES

*Du même auteur*

C. Delannoy. – **Exercices en langage C**.
N°11105, 2002, 2010 pages.

C. Delannoy. – **Le guide complet du langage C**.
N°14020, à paraître en octobre 2014, 950 pages environ.

C. Delannoy. – **S'initier à la programmation et à l'orienté objet**.
*Avec des exemples en C, C++, C#, Python, Java et PHP.*
N°14011, 2e édition, à paraître en septembre 2014, 360 pages environ.

C. Delannoy. – **Programmer en langage C++.**
N°14008, 8e édition, 2011, 820 pages (réédition avec nouvelle présentation, 2014).

C. Delannoy. – **Exercices en langage C++**.
N°12201, 3e édition, 2007, 336 pages.

C. Delannoy. – **Programmer en Java**. *Java 8*.
N°14007, 9e édition, 2014, 940 pages.

C. Delannoy. – **Exercices en Java**.
N°14009, 4e édition, 2014, 360 pages.

*Autres ouvrages*

H. Bersini, I. Wellesz. – **La programmation orientée objet**.
*Cours et exercices en UML 2 avec Java, C#, C++, Python, PHP et LinQ*.
N°13578, 6e édition, 2013, 672 pages.

B. Meyer. – **Conception et programmation orientées objet.**
N°12270, 2008, 1222 pages.

P. Roques. – **UML 2 par la pratique**
N°12565, 7e édition, 2009, 396 pages.

G. Swinnen. – **Apprendre à programmer avec Python 3**.
N°13434, 3e édition, 2012, 435 pages.

C. Blaess. – **Développement système sous Linux.**
*Ordonnancement multitâches, gestion mémoire, communications, programmation réseau*.
N°12881, 3e édition, 2011, 1004 pages.

Claude Delannoy

# Programmer en langage C

5e édition

## Cours et exercices corrigés



EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

La cinquième édition du présent ouvrage est parue en 2009 sous l'ISBN 978-2-212-11825-4. À l'occasion de ce quatrième tirage, elle bénéficie d'une nouvelle couverture.
Le texte reste inchangé, à l'exception de quelques références à la norme ISO C11 publiée en 2011





# Table des matières

# Avant-propos

Le langage C a été créé en 1972 par Denis Ritchie avec un objectif relativement limité : écrire un système d'exploitation (UNIX). Mais ses qualités opérationnelles l'ont très vite fait adopter par une large communauté de programmeurs.

Une première définition de ce langage est apparue en 1978 avec l'ouvrage de Kernighan et Ritchie *The C programming language*. Mais ce langage a continué d'évoluer après cette date à travers les différents compilateurs qui ont vu le jour. Son succès international a contribué à sa normalisation, d'abord par l'ANSI (American National Standard Institute), puis par l'ISO (International Standards Organization), plus récemment en 1993 par le CEN (Comité européen de normalisation) et enfin, en 1994, par l'AFNOR. En fait, et fort heureusement, toutes ces normes sont identiques, et l'usage veut qu'on parle de « C ANSI » ou de « C norme ANSI ».

La norme ANSI élargit, sans la contredire, la première définition de Kernighan et Ritchie. Outre la spécification de la syntaxe du langage, elle a le mérite de fournir la description d'un ensemble de fonctions qu'on doit trouver associées à tout compilateur C sous forme d'une bibliothèque standard. En revanche, compte tenu de son arrivée tardive, cette norme a cherché à « préserver l'existant », en acceptant systématiquement les anciens programmes. Elle n'a donc pas pu supprimer certaines formulations quelque peu désuètes ou redondantes. Par exemple, la première définition de Kernighan et Ritchie prévoit qu'on déclare une fonction en précisant uniquement le type de son résultat. La norme autorise qu'on la déclare sous forme d'un « prototype » (qui précise en plus le type de ses arguments) mais ne l'impose pas. Notez toutefois que le prototype deviendra obligatoire en C++.

Cet ouvrage a été conçu comme un cours de programmation en langage C. Suivant notre démarche habituelle, héritée de notre expérience de l'enseignement, nous présentons toujours les notions fondamentales sur un ou plusieurs exemples avant d'en donner plus formellement la portée générale. Souvent constitués de programmes complets, ces exemples permettent l'auto-expérimentation.

La plupart des chapitres de cet ouvrage proposent des exercices que nous vous conseillons de résoudre d'abord sur papier, en comparant vos solutions avec celles fournies en fin de volume et en réfléchissant sur les différences de rédaction qui ne manqueront pas d'apparaître. Ils serviront à la fois à contrôler les connaissances acquises et à les appliquer à des situations variées.

Nous avons cherché à privilégier tout particulièrement la clarté et la progressivité de l'exposé. Dans cet esprit, nous avons systématiquement évité les « références avant », ce qui, le cas échéant, autorise une étude séquentielle ; de même, les points les plus techniques ne sont exposés qu'une fois les bases du langage bien établies (une présentation prématurée serait perçue comme un bruit de fond masquant le fondamental).

D'une manière générale, notre fil conducteur est ce qu'on pourrait appeler le « C moderne », c'est-à-dire non pas la norme ANSI pure et dure, mais plutôt l'esprit de la norme dans ce

qu'elle a de positif. Nous pensons ainsi forger chez le lecteur de bonnes habitudes de programmation en C et, par la même occasion, nous lui facilitons son entrée future dans le monde du C++.

Enfin, outre son caractère didactique, nous avons doté cet ouvrage d'une organisation appropriée à une recherche rapide d'information :

- ses chapitres sont fortement structurés : la table des matières, fort détaillée, offre de nombreux points d'entrée,
- au fil du texte, des encadrés viennent récapituler la syntaxe des différentes instructions,
- une annexe fournit la description des fonctions les plus usitées de la bibliothèque standard (il s'agit souvent d'une reprise d'informations déjà présentées dans le texte),
- un index détaillé permet une recherche sur un point précis ; il comporte également, associé à chaque nom de fonction standard, le nom du fichier en-tête (.h) correspondant.

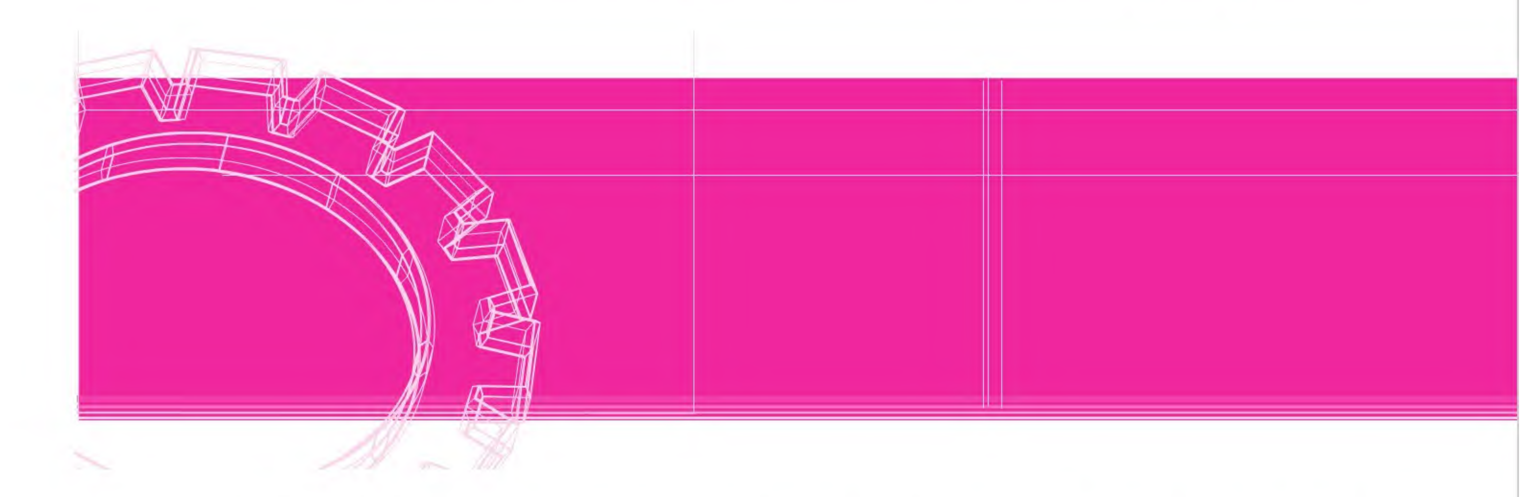
**Remarque concernant cette nouvelle édition :**

L'ISO a publié en 1999, sous la référence ISO/IEC 9899:1999, une extension de la norme du langage C, plus connue sous l'acronyme C99. Bien qu'ancienne, celle-ci est loin d'être implémentée dans sa totalité par tous les compilateurs. Dans cette nouvelle édition :

- la mention C ANSI continue à désigner l'ancienne norme, souvent baptisée C89 ou C90 ;
- lorsque cela s'est avéré justifié, nous avons précisé les nouveautés introduites par la norme C99.

# Chapitre 1

# Généralités sur le langage C



Dans ce chapitre, nous vous proposons une première approche d'un programme en langage C, basée sur deux exemples commentés. Vous y découvrirez (pour l'instant, de façon encore informelle) comment s'expriment les instructions de base (déclaration, affectation, lecture et écriture), ainsi que deux des structures fondamentales (boucle avec compteur, choix).

Nous dégagerons ensuite quelques règles générales concernant l'écriture d'un programme. Enfin, nous vous montrerons comment s'organise le développement d'un programme en vous rappelant ce que sont l'édition, la compilation, l'édition de liens et l'exécution.

## 1 Présentation par l'exemple de quelques instructions du langage C

### 1.1 Un exemple de programme en langage C

Voici un exemple de programme en langage C, accompagné d'un exemple d'exécution. Avant d'en lire les explications qui suivent, essayez d'en percevoir plus ou moins le fonctionnement.

```
#include <stdio.h>
#include <math.h>
#define NFOIS 5

main()
{ int i ;
  float x ;
  float racx ;

  printf ("Bonjour\n") ;
  printf ("Je vais vous calculer %d racines carrées\n", NFOIS) ;

  for (i=0 ; i<NFOIS ; i++)
    { printf ("Donnez un nombre : ") ;
      scanf ("%f", &x)  ;
      if (x < 0.0)
        printf ("Le nombre %f ne possède pas de racine carrée\n", x) ;
       else
        { racx = sqrt (x) ;
          printf ("Le nombre %f a pour racine carrée : %f\n", x, racx) ;
        }
    }
  printf ("Travail terminé - Au revoir") ;
}
```

```
Bonjour
Je vais vous calculer 5 racines carrées
Donnez un nombre : 4
Le nombre 4.000000 a pour racine carrée : 2.000000
Donnez un nombre : 2
Le nombre 2.000000 a pour racine carrée : 1.414214
Donnez un nombre : -3
Le nombre -3.000000 ne possède pas de racine carrée
Donnez un nombre : 5.8
Le nombre 5.800000 a pour racine carrée : 2.408319
Donnez un nombre : 12.58
Le nombre 12.580000 a pour racine carrée : 3.546829
Travail terminé - Au revoir
```

Nous reviendrons un peu plus loin sur le rôle des trois premières lignes. Pour l'instant, admettez simplement que le symbole `NFOIS` est équivalent à la valeur `5`.

## 1.2 Structure d'un programme en langage C

La ligne :

```
main()
```

se nomme un « en-tête ». Elle précise que ce qui sera décrit à sa suite est en fait le *programme principal* (main). Lorsque nous aborderons l'écriture des fonctions en C, nous verrons que celles-ci possèdent également un tel en-tête ; ainsi, en C, le programme principal apparaîtra en fait comme une fonction dont le nom (main) est imposé.

Le programme (principal) proprement dit vient à la suite de cet en-tête. Il est délimité par les accolades « { » et « } ». On dit que les instructions situées entre ces accolades forment un « bloc ». Ainsi peut-on dire que la fonction main est constituée d'un en-tête et d'un bloc ; il en ira de même pour toute fonction C. Notez qu'un bloc peut lui-même contenir d'autres blocs (c'est le cas de notre exemple). En revanche, nous verrons qu'une fonction ne peut jamais contenir d'autres fonctions.

## 1.3 Déclarations

Les trois instructions :

```
    int i ;
    float x ;
    float racx ;
```

sont des « déclarations ».

La première précise que la variable nommée i est de type int, c'est-à-dire qu'elle est destinée à contenir des nombres entiers (relatifs). Nous verrons qu'en C il existe plusieurs types d'entiers.

Les deux autres déclarations précisent que les variables x et racx sont de type float, c'est-à-dire qu'elles sont destinées à contenir des nombres flottants (approximation de nombres réels). Là encore, nous verrons qu'en C il existe plusieurs types flottants.

En C, comme dans la plupart des langages actuels, **les déclarations des types des variables sont obligatoires et doivent être regroupées au début du programme** (on devrait plutôt dire : au début de la fonction main). Il en ira de même pour toutes les variables définies dans une fonction ; on les appelle « variables locales » (en toute rigueur, les variables définies dans notre exemple sont des variables locales de la fonction main). Nous verrons également (dans le chapitre consacré aux fonctions) qu'on peut définir des variables en dehors de toute fonction ; on parlera alors de variables globales.

**Remarque C99** Suivant la norme C99, une déclaration peut figurer à n'importe quel emplacement, pour peu qu'elle apparaisse avant que la variable correspondante ne soit utilisée.

## 1.4 Pour écrire des informations : la fonction printf

L'instruction :

```
printf ("Bonjour\n") ;
```

appelle en fait une fonction prédéfinie (fournie avec le langage, et donc que vous n'avez pas à écrire vous-même) nommée `printf`. Ici, cette fonction reçoit un argument qui est :

```
"Bonjour\n"
```

Les guillemets servent à délimiter une « chaîne de caractères » (suite de caractères). La notation `\n` est conventionnelle : elle représente un caractère de fin de ligne, c'est-à-dire un caractère qui, lorsqu'il est envoyé à l'écran, provoque le passage à la ligne suivante. Nous verrons que, de manière générale, le langage C prévoit une notation de ce type (\ suivi d'un caractère) pour un certain nombre de caractères dits « de contrôle », c'est-à-dire ne possédant pas de graphisme particulier.

Notez que, apparemment, bien que `printf` soit une fonction, nous n'utilisons pas sa valeur. Nous aurons l'occasion de revenir sur ce point, propre au langage C. Pour l'instant, admettez que nous pouvons, en C, utiliser une fonction comme ce que d'autres langages nomment une « procédure » ou un « sous-programme ».

L'instruction suivante :

```
printf ("Je vais vous calculer %d racines carrées\n", NFOIS) ;
```

ressemble à la précédente avec cette différence qu'ici la fonction `printf` reçoit deux arguments. Pour comprendre son fonctionnement, il faut savoir qu'en fait le premier argument de `printf` est ce que l'on nomme un « format » ; il s'agit d'une sorte de guide qui précise comment afficher les informations qui sont fournies par les arguments suivants (le cas échéant). Ici, on demande à `printf` d'afficher suivant ce format :

```
"Je vais vous calculer %d racines carrées\n"
```

la valeur de `NFOIS`, c'est-à-dire, la valeur `5`.

Ce format est, comme précédemment, une chaîne de caractères. Toutefois, vous constatez la présence d'un caractère `%`. Celui-ci signifie que le caractère suivant est, non plus du texte à afficher tel quel, mais un « code de format ». Ce dernier précise qu'il faut considérer la valeur reçue (en argument suivant, donc ici `5`) comme un entier et l'afficher en décimal. Notez bien que tout ce qui, dans le format, n'est pas un code de format, est affiché tel quel ; il en va ainsi du texte « `racines carrées\n` ».

Il peut paraître surprenant d'avoir à spécifier à nouveau dans le code format que `NFOIS(5)` est un entier alors que l'on pourrait penser que le compilateur est bien capable de s'en apercevoir (quoiqu'il ne puisse pas deviner que nous voulons l'écrire en décimal et non pas, par exemple, en hexadécimal). Nous aurons l'occasion de revenir sur ce phénomène dont l'explication réside

essentiellement dans le fait que `printf` est une fonction, autrement dit que les instructions correspondantes seront incorporées, non pas à la compilation, mais lors de l'édition de liens.

Cependant, dès maintenant, **sachez qu'il vous faudra toujours veiller à accorder le code de format au type de la valeur correspondante**. Si vous ne respectez pas cette règle, vous risquez fort d'afficher des valeurs totalement fantaisistes.

## 1.5 Pour faire une répétition : l'instruction `for`

Comme nous le verrons, en langage C, il existe plusieurs façons de réaliser une répétition (on dit aussi une « boucle »). Ici, nous avons utilisé l'instruction `for` :

```
   for (i=0 ; i<NFOIS ; i++)
```

Son rôle est de répéter le bloc (délimité par des accolades « `{` » et « `}` ») figurant à sa suite, en respectant les consignes suivantes :

- avant de commencer cette répétition, réaliser :

```
i = 0
```

- avant chaque nouvelle exécution du bloc (tour de boucle), examiner la condition :

```
i < NFOIS
```

si elle est satisfaite, exécuter le bloc indiqué, sinon passer à l'instruction suivant ce bloc : à la fin de chaque exécution du bloc, réaliser :

```
 i++
```

Il s'agit là d'une notation propre au langage C qui est équivalente à :

```
i = i + 1
```

En définitive, vous voyez qu'ici notre bloc sera répété cinq fois.

## 1.6 Pour lire des informations : la fonction `scanf`

La première instruction du bloc répété par l'instruction `for` affiche simplement le message `Donnez un nombre:`. Notez qu'ici nous n'avons pas prévu de changement de ligne à la fin. La seconde instruction du bloc :

```
scanf ("%f", &x) ;
```

est un appel de la fonction prédéfinie `scanf` dont le rôle est de lire une information au clavier. Comme `printf`, la fonction `scanf` possède en premier argument un format exprimé sous forme d'une chaîne de caractères, ici :

```
"%f"
```

ce qui correspond à une valeur flottante (plus tard, nous verrons précisément sous quelle forme elle peut être fournie ; l'exemple d'exécution du programme vous en donne déjà une bonne idée !). Notez bien qu'ici, contrairement à ce qui se produisait pour `printf`, nous n'avons aucune raison de trouver, dans ce format, d'autres caractères que ceux qui servent à définir un code de format.

Comme nous pouvons nous y attendre, les arguments (ici, il n'y en a qu'un) précisent dans quelles variables on souhaite placer les valeurs lues. Il est fort probable que vous vous attendiez à trouver simplement `x` et non pas `&x`.

En fait, la nature même du langage C fait qu'une telle notation reviendrait à transmettre à la fonction `scanf` la **valeur** de la variable `x` (laquelle, d'ailleurs, n'aurait pas encore reçu de valeur précise). Or, manifestement, la fonction `scanf` doit être en mesure de ranger la valeur qu'elle aura lue dans l'emplacement correspondant à cette variable, c'est-à-dire à son **adresse**. Effectivement, nous verrons que `&` est un opérateur signifiant *adresse de*.

Notez bien que si, par mégarde, vous écrivez `x` au lieu de `&x`, le compilateur ne détectera pas d'erreur. Au moment de l'exécution, `scanf` prendra l'information reçue en deuxième argument (valeur de `x`) pour une adresse à laquelle elle rangera la valeur lue. Cela signifie qu'on viendra tout simplement écraser un emplacement indéterminé de la mémoire ; les conséquences pourront alors être quelconques.

## 1.7 Pour faire des choix : l'instruction `if`

Les lignes :

```
if (x < 0.0)
   printf ("Le nombre %f ne possède pas de racine carrée\n", x) ;
 else
   { racx = sqrt (x) ;
     printf ("Le nombre %f a pour racine carrée : %f\n", x, racx) ;
   }
```

constituent une instruction de choix basée sur la condition `x < 0.0`. Si cette condition est vraie, on exécute l'instruction suivante, c'est-à-dire :

```
   printf ("Le nombre %f ne possède pas de racine carrée\n", x) ;
```

Si elle est fausse, on exécute l'instruction suivant le mot `else`, c'est-à-dire, ici, le bloc :

```
   { racx = sqrt (x) ;
     printf ("Le nombre %f a pour racine carrée : %f\n", x, racx) ;
   }
```

Notez qu'il existe un mot `else` mais pas de mot `then`. La syntaxe de l'instruction `if` (notamment grâce à la présence de parenthèses qui encadrent la condition) le rend inutile.

La fonction `sqrt` fournit la valeur de la racine carrée d'une valeur flottante qu'on lui transmet en argument.

**Remarque**

Une instruction telle que :

```
racx = sqrt (x) ;
```

est une instruction classique d'affectation : elle donne à la variable `racx` la valeur de l'expression située à droite du signe égal. Nous verrons plus tard qu'en C l'affectation peut prendre des formes plus élaborées.

Notez que C dispose de trois sortes d'instructions :

- des instructions simples, terminées obligatoirement par un point-virgule,
- des instructions de structuration telles que `if` ou `for`,
- des blocs (délimités par { et }).

Les deux dernières ont une définition « récursive » puisqu'elles peuvent contenir, à leur tour, n'importe laquelle des trois formes.

Lorsque nous parlerons d'instruction, sans précisions supplémentaires, il pourra s'agir de n'importe laquelle des trois formes ci-dessus.

## 1.8 Les directives à destination du préprocesseur

Les trois premières lignes de notre programme :

```
#include <stdio.h>
#include <math.h>
#define NFOIS 5
```

sont en fait un peu particulières. Il s'agit de directives qui seront prises en compte avant la traduction (compilation) du programme. Ces directives, contrairement au reste du programme, doivent être écrites à raison d'une par ligne et elles doivent obligatoirement commencer en début de ligne. Leur emplacement au sein du programme n'est soumis à aucune contrainte (mais une directive ne s'applique qu'à la partie du programme qui lui succède). D'une manière générale, il est préférable de les placer au début, comme nous l'avons fait ici.

Les deux premières directives demandent en fait d'introduire (avant compilation) des instructions (en langage C) situées dans les fichiers `stdio.h` et `math.h`. Leur rôle ne sera complètement compréhensible qu'ultérieurement.

Pour l'instant, notez que, dès lors que vous faites appel à une fonction prédéfinie, il est nécessaire d'incorporer de tels fichiers, nommés « fichiers en-têtes », qui contiennent des déclarations appropriées concernant cette fonction : `stdio.h` pour `printf` et `scanf`, `math.h` pour `sqrt`. Fréquemment, ces déclarations permettront au compilateur d'effectuer des contrôles sur le nombre et le type des arguments que vous mentionnerez dans l'appel de votre fonction.

Notez qu'un même fichier en-tête contient des déclarations relatives à plusieurs fonctions. En général, il est indispensable d'incorporer stdio.h.

La troisième directive demande simplement de remplacer systématiquement, dans toute la suite du programme, le symbole NFOIS par 5. Autrement dit, le programme qui sera réellement compilé comportera ces instrutions :

```
printf ("Je vais vous calculer %d racines carrées\n", 5) ;

for (i=0 ; i<5 ; i++)
```

Notez toutefois que le programme proposé est plus facile à adapter lorsque l'on emploie une directive define.

**Remarque**

**Important :** Dans notre exemple, la directive #define servait à définir la valeur d'un symbole. Nous verrons (dans le chapitre consacré au préprocesseur) que cette directive sert également à définir ce que l'on nomme une « macro ». Une macro s'utilise comme une fonction ; en particulier, elle peut posséder des arguments. Mais le préprocesseur remplacera chaque appel par la ou les instructions C correspondantes. Dans le cas d'une (vraie) fonction, une telle substitution n'existe pas ; au contraire, c'est l'éditeur de liens qui incorporera (une seule fois quel que soit le nombre d'appels) les instructions machine correspondantes.

## 1.9 Un second exemple de programme

Voici un second exemple de programme destiné à vous montrer l'utilisation du type « caractère ». Il demande à l'utilisateur de choisir une opération parmi l'addition ou la multiplication, puis de fournir deux nombres entiers ; il affiche alors le résultat correspondant.

```
#include <stdio.h>
main()
{
   char op ;
   int n1, n2 ;
   printf ("opération souhaitée (+ ou *) ? ") ;
   scanf ("%c", &op) ;
   printf ("donnez 2 nombres entiers : ") ;
   scanf ("%d %d", &n1, &n2) ;
   if (op == '+') printf ("leur somme  est : %d ", n1+n2) ;
         else  printf ("leur produit est : %d ", n1*n2) ;
}
```

Ici, nous déclarons que la variable `op` est de type caractère (`char`). Une telle variable est destinée à contenir un caractère quelconque (codé, bien sûr, sous forme binaire !).

L'instruction `scanf ("%c", &op)` permet de lire un caractère au clavier et de le ranger dans `op`. Notez le code `%c` correspondant au type `char` (n'oubliez pas le `&` devant `op`). L'instruction `if` permet d'afficher la somme ou le produit de deux nombres, suivant le caractère contenu dans `op`. Notez que :

- la relation d'égalité se traduit par le signe == (et non = qui représente l'affectation et qui, ici, comme nous le verrons plus tard, serait admis mais avec une autre signification !).
- la notation '+' représente une constante caractère. Notez bien que C n'utilise pas les mêmes délimiteurs pour les chaînes (il s'agit de ") et pour les caractères.

Remarquez que, tel qu'il a été écrit, notre programme calcule le produit, dès lors que le caractère fourni par l'utilisateur n'est pas +.

**Remarques**

On pourrait penser à inverser l'ordre des deux instructions de lecture en écrivant :

```
scanf ("%d %d", &n1, &n2) ;
   ...
scanf ("%c", &op) ;
```

Toutefois, dans ce cas, une petite difficulté apparaîtrait : le caractère lu par le second appel de `scanf` serait toujours différent de + (ou de *). Il s'agirait en fait du caractère de fin de ligne `\n` (fourni par la validation de la réponse précédente). Le mécanisme exact vous sera expliqué dans le chapitre relatif aux « entrées-sorties conversationnelles » ; pour l'instant, sachez que vous pouvez régler le problème en effectuant une lecture d'un caractère supplémentaire.

Au lieu de :

```
scanf ("%d", &op) ;
```

on pourrait écrire :

```
op = getchar () ;
```

Cette instruction affecterait à la variable `op` le résultat fourni par la fonction `getchar` (qui ne reçoit aucun argument - n'omettez toutefois pas les parenthèses !).

D'une manière générale, il existe une fonction symétrique `putchar` ; ainsi :

```
putchar (op) ;
```

affiche le caractère contenu dans `op`.

Notez que généralement `getchar` et `putchar` sont, non pas des vraies fonctions, mais des macros dont la définition figure dans `stdio.h`.

# 2 Quelques règles d'écriture

Ce paragraphe expose un certain nombre de règles générales intervenant dans l'écriture d'un programme en langage C. Nous y parlerons précisément de ce que l'on appelle les « identificateurs » et les « mots-clés », du format libre dans lequel on écrit les instructions, ainsi que de l'usage des séparateurs et des commentaires.

## 2.1 Les identificateurs

Les identificateurs servent à désigner les différents « objets » manipulés par le programme : variables, fonctions, etc. (Nous rencontrerons ultérieurement les autres objets manipulés par le langage C : constantes, étiquettes de structure, d'union ou d'énumération, membres de structure ou d'union, types, étiquettes d'instruction `GOTO`, macros). Comme dans la plupart des langages, ils sont formés d'une suite de caractères choisis parmi les **lettres** ou les **chiffres**, le premier d'entre eux étant nécessairement une lettre.

En ce qui concerne les lettres :

- le caractère souligné (_) est considéré comme une lettre. Il peut donc apparaître au début d'un identificateur. Voici quelques identificateurs corrects :

  ```
  lg_lig    valeur_5   _total   _89
  ```

- les majuscules et les minuscules sont autorisées mais ne sont pas équivalentes. Ainsi, en C, les identificateurs *ligne* et *Ligne* désignent deux objets différents.

En ce qui concerne la longueur des identificateurs, la norme ANSI prévoit qu'au moins les 31 premiers caractères soient « significatifs » (autrement dit, deux identificateurs qui diffèrent par leurs 31 premières lettres désigneront deux objets différents).

## 2.2 Les mots-clés

Certains « mots-clés » sont réservés par le langage à un usage bien défini et ne peuvent pas être utilisés comme identificateurs. En voici la liste, classée par ordre alphabétique.

*Les mots-clés du langage C*

| | | | | | |
|---|---|---|---|---|---|
| auto | default | float | register | struct | volatile |
| break | do | for | return | switch | while |
| case | double | goto | short | typedef | |
| char | else | if | signed | union | |
| const | enum | int | sizeof | unsigned | |
| continue | extern | long | static | void | |

## 2.3 Les séparateurs

Dans notre langue écrite, les différents mots sont séparés par un espace, un signe de ponctuation ou une fin de ligne.

Il en va quasiment de même en langage C dans lequel les règles vont donc paraître naturelles. Ainsi, dans un programme, deux identificateurs successifs entre lesquels la syntaxe n'impose aucun signe particulier (tel que : `, = ; * ( ) [ ] { }`) doivent impérativement être séparés soit par un espace, soit par une fin de ligne. En revanche, dès que la syntaxe impose un séparateur quelconque, il n'est alors pas nécessaire de prévoir d'espaces supplémentaires (bien qu'en pratique cela améliore la lisibilité du programme).

Ainsi, vous devrez impérativement écrire :

```
int x,y
```

et non :

```
intx,y
```

En revanche, vous pourrez écrire indifféremment :

```
int n,compte,total,p
```

ou plus lisiblement :

```
int n, compte, total, p
```

## 2.4 Le format libre

Le langage C autorise une mise en page parfaitement libre. En particulier, une instruction peut s'étendre sur un nombre quelconque de lignes, et une même ligne peut comporter autant d'instructions que vous le souhaitez. Les fins de ligne ne jouent pas de rôle particulier, si ce n'est celui de séparateur, au même titre qu'un espace, sauf dans les « constantes chaînes » où elles sont interdites ; de telles constantes doivent impérativement être écrites à l'intérieur d'une seule ligne. Un identificateur ne peut être coupé en deux par une fin de ligne, ce qui semble évident.

Bien entendu, cette liberté de mise en page possède des contreparties. Notamment, le risque existe, si l'on n'y prend garde, d'aboutir à des programmes peu lisibles.

À titre d'exemple, voyez comment pourrait être (mal) présenté notre programme précédent :

*Exemple de programme mal présenté*

```
#include <stdio.h>
#include <math.h>
#define NFOIS 5
main() {  int i ;  float
       x
  ; float racx ; printf ("Bonjour\n") ; printf
     ("Je vais vous calculer %d racines carrées\n", NFOIS) ;  for (i=
    0 ; i<NFOIS ; i++) { printf ("Donnez un nombre : ") ; scanf ("%f"
     , &x) ;  if (x < 0.0)
   printf ("Le nombre %f ne possède pas de racine carrée\n", x) ;  else
 { racx = sqrt (x) ; printf ("Le nombre %f a pour racine carrée : %f\n",
       x, racx) ;  }     }   printf ("Travail terminé - Au revoir") ;}
```

## 2.5 Les commentaires

Comme tout langage évolué, le langage C autorise la présence de commentaires dans vos programmes source. Il s'agit de textes explicatifs destinés aux lecteurs du programme et qui n'ont aucune incidence sur sa compilation.

Ils sont formés de caractères quelconques placés entre les symboles `/*` et `*/`. Ils peuvent apparaître à tout endroit du programme où un espace est autorisé. En général, cependant, on se limitera à des emplacements propices à une bonne lisibilité du programme.

Voici quelques exemples de commentaires :

```
/* programme de calcul de racines carrées */

/* commentaire fantaisiste &ç§{<>} ?%!!!!!! */

/* commentaire s'étendant
   sur plusieurs lignes
   de programme source    */

/* ============================================
   *    commentaire quelque peu esthétique    *
   *    et encadré, pouvant servir,        *
   *    par exemple, d'en-tête de programme   *
   ============================================ */
```



Voici un exemple de commentaires qui, situés au sein d'une instruction de déclaration, permettent de définir le rôle des différentes variables :

```
int i ;               /* compteur de boucle */
float x ;             /* nombre dont on veut la racine carrée */
float racx ;          /* racine carrée du nombre */
```

**Remarque C99** La norme C99 autorise une seconde forme de commentaire, dit « de fin de ligne », que l'on retrouve également en C++. Un tel commentaire est introduit par // et tout ce qui suit ces deux caractères jusqu'à la fin de la ligne est considéré comme un commentaire. En voici un exemple :

```
printf ("bonjour\n") ;   // formule de politesse
```

# 3 Création d'un programme en langage C

La manière de développer et d'utiliser un programme en langage C dépend naturellement de l'environnement de programmation dans lequel vous travaillez. Nous vous fournissons ici quelques indications générales (s'appliquant à n'importe quel environnement) concernant ce que l'on pourrait appeler les grandes étapes de la création d'un programme, à savoir : édition du programme, compilation et édition de liens.

## 3.1 L'édition du programme

L'édition du programme (on dit aussi parfois « saisie ») consiste à créer, à partir d'un clavier, tout ou partie du texte d'un programme qu'on nomme « programme source ». En général, ce texte sera conservé dans un fichier que l'on nommera « fichier source ».

Chaque système possède ses propres conventions de dénomination des fichiers. En général, un fichier peut, en plus de son nom, être caractérisé par un groupe de caractères (au moins 3) qu'on appelle une « extension » (ou, parfois un « type ») ; la plupart du temps, en langage C, les fichiers source porteront l'extension C.

## 3.2 La compilation

Elle consiste à traduire le programme source (ou le contenu d'un fichier source) en langage machine, en faisant appel à un programme nommé compilateur. En langage C, compte tenu de l'existence d'un préprocesseur, cette opération de compilation comporte en fait deux étapes :

- **traitement par le préprocesseur** : ce dernier exécute simplement les directives qui le concernent (il les reconnaît au fait qu'elles commencent par un caractère #). Il produit, en résultat, un programme source en langage C pur. Notez bien qu'il s'agit toujours d'un vrai texte, au même titre qu'un programme source : la plupart des environnements de programmation vous permettent d'ailleurs, si vous le souhaitez, de connaître le résultat fourni par le préprocesseur.

- **compilation** proprement dite, c'est-à-dire traduction en langage machine du texte en langage C fourni par le préprocesseur.

Le résultat de la compilation porte le nom de module objet.

## 3.3 L'édition de liens

Le module objet créé par le compilateur n'est pas directement exécutable. Il lui manque, au moins, les différents modules objet correspondant aux fonctions prédéfinies (on dit aussi « fonctions standard ») utilisées par votre programme (comme `printf`, `scanf`, `sqrt`).

C'est effectivement le rôle de l'éditeur de liens que d'aller rechercher dans la bibliothèque standard les modules objet nécessaires. Notez que cette bibliothèque est une collection de modules objet organisée, suivant l'implémentation concernée, en un ou plusieurs fichiers.

Le résultat de l'édition de liens est ce que l'on nomme un programme exécutable, c'est-à-dire un ensemble autonome d'instructions en langage machine. Si ce programme exécutable est rangé dans un fichier, il pourra ultérieurement être exécuté sans qu'il soit nécessaire de faire appel à un quelconque composant de l'environnement de programmation en C.

## 3.4 Les fichiers en-tête

Nous avons vu que, grâce à la directive `#include`, vous pouviez demander au préprocesseur d'introduire des instructions (en langage C) provenant de ce que l'on appelle des fichiers « en-tête ». De tels fichiers comportent, entre autres choses :

- des déclarations relatives aux fonctions prédéfinies,
- des définitions de macros prédéfinies.

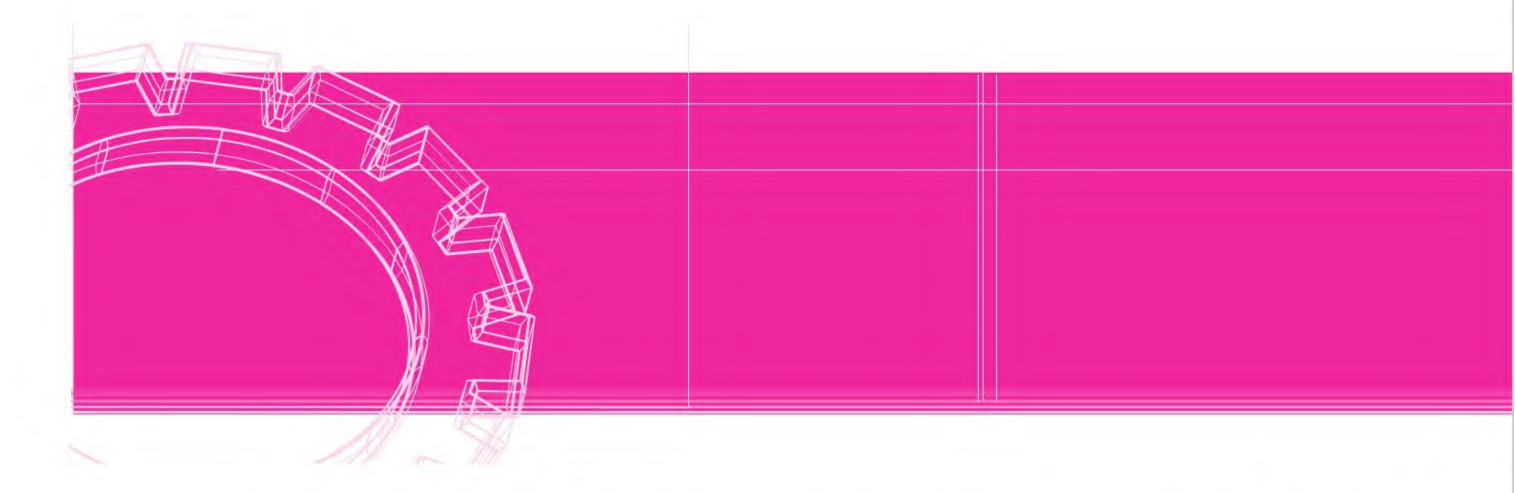
Lorsqu'on écrit un programme, on ne fait pas toujours la différence entre fonction et macro, puisque celles-ci s'utilisent de la même manière. Toutefois, les fonctions et les macros sont traitées de façon totalement différente par l'ensemble « préprocesseur + compilateur + éditeur de liens ».

En effet, les appels de macros sont remplacés (par du C) par le préprocesseur, du moins si vous avez incorporé le fichier en-tête correspondant. Si vous ne l'avez pas fait, aucun remplacement ne sera effectué, mais aucune erreur de compilation ne sera détectée : le compilateur croira simplement avoir affaire à un appel de fonction ; ce n'est que l'éditeur de liens qui, ne la trouvant pas dans la bibliothèque standard, vous fournira un message.

Les fonctions, quant à elles, sont incorporées par l'éditeur de liens. Cela reste vrai, même si vous omettez la directive `#include` correspondante ; dans ce cas, simplement, le compilateur n'aura pas disposé d'informations appropriées permettant d'effectuer des contrôles d'arguments (nombre et type) et de mettre en place d'éventuelles conversions ; aucune erreur ne sera signalée à la compilation ni à l'édition de liens ; les conséquences n'apparaîtront que lors de l'exécution : elles peuvent être invisibles dans le cas de fonctions comme `printf` ou, au contraire, conduire à des résultats erronés dans le cas de fonctions comme `sqrt`.

# Chapitre 2

# Les types de base du langage C



Les types `char`, `int` et `float` que nous avons déjà rencontrés sont souvent dits « scalaires » ou « simples », car, à un instant donné, une variable d'un tel type contient une seule valeur. Ils s'opposent aux types « structurés » (on dit aussi « agrégés ») qui correspondent à des variables qui, à un instant donné, contiennent plusieurs valeurs (de même type ou non). Ici, nous étudierons en détail ce que l'on appelle les **types de base** du langage C ; il s'agit des types scalaires à partir desquels pourront être construits tous les autres, dits « types dérivés », qu'il s'agisse :

- de types structurés comme les tableaux, les structures ou les unions,
- d'autres types simples comme les pointeurs ou les énumérations.

Auparavant, cependant, nous vous proposons de faire un bref rappel concernant la manière dont l'information est représentée dans un ordinateur et la notion de type qui en découle.

## 1 La notion de type

La mémoire centrale est un ensemble de positions binaires nommées bits. Les bits sont regroupés en octets (8 bits), et chaque octet est repéré par ce qu'on nomme son adresse.

L'ordinateur, compte tenu de sa technologie actuelle, ne sait représenter et traiter que des informations exprimées sous forme binaire. Toute information, quelle que soit sa nature, devra être **codée** sous cette forme. Dans ces conditions, on voit qu'il ne suffit pas de connaître le contenu d'un emplacement de la mémoire (d'un ou de plusieurs octets) pour être en mesure de lui attribuer une signification. Par exemple, si vous savez qu'un octet contient le « motif binaire » suivant :

```
01001101
```

vous pouvez considérer que cela représente le nombre entier `77` (puisque le motif ci-dessus correspond à la représentation en base 2 de ce nombre). Mais pourquoi cela représenterait-il un nombre ? En effet, toutes les informations (nombres entiers, nombres réels, nombres complexes, caractères, instructions de programme en langage machine, graphiques...) devront, au bout du compte, être codées en binaire.

Dans ces conditions, les huit bits ci-dessus peuvent peut-être représenter un caractère ; dans ce cas, si nous connaissons la convention employée sur la machine concernée pour représenter les caractères, nous pouvons lui faire correspondre un caractère donné (par exemple M, dans le cas du code ASCII). Ils peuvent également représenter une partie d'une instruction machine ou d'un nombre entier codé sur deux octets, ou d'un nombre réel codé sur 4 octets, ou...

On comprend donc qu'il n'est pas possible d'attribuer une signification à une information binaire tant que l'on ne connaît pas la manière dont elle a été codée. Qui plus est, en général, il ne sera même pas possible de « traiter » cette information. Par exemple, pour additionner deux informations, il faudra savoir quel codage a été employé afin de pouvoir mettre en œuvre les bonnes instructions (en langage machine). Par exemple, on ne fait pas appel aux mêmes circuits électroniques pour additionner deux nombres codés sous forme « entière » et deux nombres codés sous forme « flottante ».

D'une manière générale, la notion de type, telle qu'elle existe dans les langages évolués, sert à régler (entre autres choses) les problèmes que nous venons d'évoquer.

Les types de base du langage C se répartissent en trois grandes catégories en fonction de la nature des informations qu'ils permettent de représenter :

- nombres entiers (mot-clé **`int`**),
- nombres flottants (mot-clé **`float`** ou **`double`**),
- caractères (mot-clé **`char`**) ; nous verrons qu'en fait `char` apparaît (en C) comme un cas particulier de `int`.



# 2 Les types entiers

## 2.1 Leur représentation en mémoire

Le mot-clé `int` correspond à la représentation de nombres entiers relatifs. Pour ce faire : un bit est réservé pour représenter le signe du nombre (en général `0` correspond à un nombre

positif) ; les autres bits servent à représenter la valeur absolue du nombre (en toute rigueur, on la représente sous la forme de ce que l'on nomme le « complément à deux ». Nous y reviendrons dans le chapitre 13).

## 2.2 Les différents types d'entiers

Le C prévoit que, sur une machine donnée, on puisse trouver jusqu'à trois tailles différentes d'entiers, désignées par les mots-clés suivants :

- **`short int`** (qu'on peut abréger en `short`),
- **`int`** (c'est celui que nous avons rencontré dans le chapitre précédent),
- **`long int`** (qu'on peut abréger en `long`).

Chaque taille impose naturellement ses limites. Toutefois, ces dernières dépendent, non seulement du mot-clé considéré, mais également de la machine utilisée : tous les `int` n'ont pas la même taille sur toutes les machines ! Fréquemment, deux des trois mots-clés correspondent à une même taille (par exemple, sur PC, `short` et `int` correspondent à 16 bits, tandis que `long` correspond à 32 bits).

À titre indicatif, avec 16 bits, on représente des entiers s'étendant de `-32 768` à `32 767` ; avec 32 bits, on peut couvrir les valeurs allant de `-2 147 483 648` à `2 147 483 647`.

**Remarque**

En toute rigueur, chacun des trois types (`short`, `int` et `long`) peut être nuancé par l'utilisation du qualificatif `unsigned` (non signé). Dans ce cas, il n'y a plus de bit réservé au signe et on ne représente plus que des nombres positifs. Son emploi est réservé à des situations particulières. Nous y reviendrons dans le chapitre 13.

**Remarque C99**

La norme C99 introduit le type `long long`, ainsi que des types permettant de choisir :

- soit la taille correspondante, par exemple `int16` pour des entiers codés sur 16 bits ou `int32` pour des entiers codés sur 32 bits ;
- soit une taille minimale, par exemple `int_least32_t` pour un entier d'au moins 32 bits.

## 2.3 Notation des constantes entières

La façon la plus naturelle d'introduire une constante entière dans un programme est de l'écrire simplement sous forme décimale, avec ou sans signe, comme dans ces exemples :

```
+533    48    -273
```

Il est également possible d'utiliser une notation octale ou hexadécimale. Nous en reparlerons dans le chapitre 13.

# 3 Les types flottants

## 3.1 Les différents types et leur représentation en mémoire

Les types flottants permettent de représenter, **de manière approchée**, une partie des nombres réels. Pour ce faire, ils s'inspirent de la notation scientifique (ou exponentielle) bien connue qui consiste à écrire un nombre sous la forme 1.5 $10^{22}$ ou 0.472 $10^{-8}$ ; dans une telle notation, on nomme « mantisses » les quantités telles que `1.5` ou `0.472` et « exposants » les quantités telles que `22` ou `-8`.

Plus précisément, un nombre réel sera représenté en flottant en déterminant deux quantités M (mantisse) et E (exposant) telles que la valeur

$$M \, . \, B^{E}$$

représente une approximation de ce nombre. La base B est généralement unique pour une machine donnée (il s'agit souvent de 2 ou de 16) et elle ne figure pas explicitement dans la représentation machine du nombre.

Le C prévoit trois types de flottants correspondant à des tailles différentes : `float`, `double` et `long double`. La connaissance des caractéristiques exactes du système de codage n'est généralement pas indispensable, sauf lorsque l'on doit faire une analyse fine des erreurs de calcul. En revanche, il est important de noter que de telles représentations sont caractérisées par deux éléments :

- *la précision* : lors du codage d'un nombre décimal quelconque dans un type flottant, il est nécessaire de ne conserver qu'un nombre fini de bits. Or la plupart des nombres s'exprimant avec un nombre limité de décimales ne peuvent pas s'exprimer de façon exacte dans un tel codage. On est donc obligé de se limiter à une représentation approchée en faisant ce que l'on nomme une erreur de troncature. Quelle que soit la machine utilisée, on est assuré que cette erreur (relative) ne dépassera pas $10^{-6}$ pour le type `float` et $10^{-10}$ pour le type `long double`.
- *le domaine couvert*, c'est-à-dire l'ensemble des nombres représentables à l'erreur de troncature près. Là encore, quelle que soit la machine utilisée, on est assuré qu'il s'étendra au moins de $10^{-37}$ à $10^{+37}$.

## 3.2 Notation des constantes flottantes

Comme dans la plupart des langages, les constantes flottantes peuvent s'écrire indifféremment suivant l'une des deux notations :

- décimale,
- exponentielle.

La notation décimale doit comporter obligatoirement un point (correspondant à notre virgule). La partie entière ou la partie décimale peut être omise (mais, bien sûr, pas toutes les deux en même temps !). En voici quelques exemples corrects :

```
12.43    -0.38    -.38    4.    .27
```

En revanche, la constante `47` serait considérée comme entière et non comme flottante. Dans la pratique, ce fait aura peu d'importance, si ce n'est au niveau du temps d'exécution, compte tenu des conversions automatiques qui seront mises en place par le compilateur (et dont nous parlerons dans le chapitre suivant).

La notation exponentielle utilise la lettre `e` (ou `E`) pour introduire un exposant entier (puissance de 10), avec ou sans signe. La mantisse peut être n'importe quel nombre décimal ou entier (le point peut être absent dès que l'on utilise un exposant). Voici quelques exemples corrects (les exemples d'une même ligne étant équivalents) :

```
    4.25E4         4.25e+4       42.5E3
54.27E-32       542.7E-33    5427e-34
      48e13           48.e13      48.0E13
```

Par défaut, toutes les constantes sont créées par le compilateur dans le type `double`. Il est toutefois possible d'imposer à une constante flottante :

- d'être du type `float`, en faisant suivre son écriture de la lettre `F` (ou `f`) : cela permet de gagner un peu de place mémoire, en contrepartie d'une éventuelle perte de précision (le gain en place et la perte en précision dépendant de la machine concernée).
- d'être du type `long double`, en faisant suivre son écriture de la lettre `L` (ou `l`) : cela permet de gagner éventuellement en précision, en contrepartie d'une perte de place mémoire (le gain en précision et la perte en place dépendant de la machine concernée).

# 4 Les types caractères

## 4.1 La notion de caractère en langage C

Comme la plupart des langages, C permet de manipuler des caractères codés en mémoire sur un octet. Bien entendu, le code employé, ainsi que l'ensemble des caractères représentables, dépend de l'environnement de programmation utilisé (c'est-à-dire à la fois de la machine concernée et du compilateur employé). Néanmoins, on est toujours certain de disposer des lettres (majuscules et minuscules), des chiffres, des signes de ponctuation et des différents séparateurs (en fait, tous ceux que l'on emploie pour écrire un programme !). En revanche, les caractères nationaux (caractères accentués ou ç) ou les caractères semi-graphiques ne figurent pas dans tous les environnements.

Par ailleurs, la notion de caractère en C dépasse celle de caractère imprimable, c'est-à-dire auquel est obligatoirement associé un graphisme (et qu'on peut donc imprimer ou afficher sur un écran). C'est ainsi qu'il existe certains caractères de changement de ligne, de tabulation, d'activation d'une alarme sonore (cloche),... Nous avons d'ailleurs déjà utilisé le premier (sous la forme `\n`).

**Remarque**

De tels caractères sont souvent nommés « caractères de contrôle ». Dans le code ASCII (restreint ou non), ils ont des codes compris entre 0 et 31.

## 4.2 Notation des constantes caractères

Les constantes de type « caractère », lorsqu'elles correspondent à des caractères imprimables, se notent de façon classique, en écrivant entre apostrophes (ou quotes) le caractère voulu, comme dans ces exemples :

```
'a'      'Y'      '+'      '$'
```

Certains caractères non imprimables possèdent une représentation conventionnelle utilisant le caractère « \ », nommé « antislash » (en anglais, il se nomme « back-slash », en français, on le nomme aussi « barre inverse » ou « contre-slash »). Dans cette catégorie, on trouve également quelques caractères (`\`, `'`, `"` et `?`) qui, bien que disposant d'un graphisme, jouent un rôle particulier de délimiteur qui les empêche d'être notés de manière classique entre deux apostrophes.

Voici la liste de ces caractères.

*Caractères disposant d'une notation spéciale*

| NOTATION EN C | CODE ASCII (hexadécimal) | ABRÉVIATION USUELLE | SIGNIFICATION |
|---|---|---|---|
| \a | 07 | BEL | cloche ou bip (alert ou audible bell) |
| \b | 08 | BS | Retour arrière (Backspace) |
| \f | 0C | FF | Saut de page (Form Feed) |
| \n | 0A | LF | Saut de ligne (Line Feed) |
| \r | 0D | CR | Retour chariot (Carriage Return) |
| \t | 09 | HT | Tabulation horizontale (Horizontal Tab) |
| \v | 0B | VT | Tabulation verticale (Vertical Tab) |
| \\ | 5C | \ | |
| \' | 2C | ' | |
| \" | 22 | " | |
| \? | 3F | ? | |

De plus, il est possible d'utiliser directement le code du caractère en l'exprimant, toujours à la suite du caractère « antislash » :

- soit sous forme **octale**,
- soit sous forme **hexadécimale** précédée de **x**.

Voici quelques exemples de notations équivalentes, dans le code ASCII :

```
'A'     '\x41'   '\101'
'\r'    '\x0d'   '\15'    '\015'
'\a'    '\x07'   '\x7'    '\07'    '\007'
```

**Remarques**

En fait, il existe plusieurs versions de code ASCII, mais toutes ont en commun la première moitié des codes (correspondant aux caractères qu'on trouve dans toutes les implémentations) ; les exemples cités ici appartiennent bien à cette partie commune.

Le caractère `\`, suivi d'un caractère autre que ceux du tableau ci-dessus ou d'un chiffre de `0` à `7` est simplement ignoré. Ainsi, dans le cas où l'on a affaire au code ASCII, `\9` correspond au caractère `9` (de code ASCII 57), tandis que `\7` correspond au caractère de code ASCII 7, c'est-à-dire la « cloche ».

En fait, la norme prévoit deux types : `signed char` et `unsigned char` (`char` correspondant soit à l'un, soit à l'autre, suivant le compilateur utilisé). Là encore, nous y reviendrons dans le chapitre 13. Pour l'instant, sachez que cet attribut de signe n'agit pas sur la représentation d'un caractère en mémoire. En revanche, il pourra avoir un rôle dans le cas où l'on s'intéresse à la valeur numérique associée à un caractère.

# 5 Initialisation et constantes

1. Nous avons déjà vu que la directive `#define` permettait de donner une valeur à un symbole. Dans ce cas, le préprocesseur effectue le remplacement correspondant avant la compilation.
2. Par ailleurs, il est possible d'initialiser une variable lors de sa déclaration comme dans :

   ```
   int n = 15 ;
   ```

   Ici, pour le compilateur, `n` est une **variable** de type `int` dans laquelle il placera la valeur `15` ; mais rien n'empêche que cette valeur initiale évolue lors de l'exécution du programme. Notez d'ailleurs que la déclaration précédente pourrait être remplacée par une déclaration ordinaire (`int n`), suivie un peu plus loin d'une affectation (`n=15`) ; la seule différence résiderait dans l'instant où `n` recevrait la valeur `15`.

3. En fait, il est possible de déclarer que la valeur d'une variable ne doit pas changer lors de l'exécution du programme. Par exemple, avec :

```
const int n = 20 ;
```

on déclare `n` de type `int` et de valeur (initiale) `20` mais, de surcroît, les éventuelles instructions modifiant la valeur de `n` seront rejetées par le compilateur.

On pourrait penser qu'une déclaration (par `const`) remplace avantageusement l'emploi de `define`. En fait, nous verrons que les choses ne sont pas aussi simples, car les variables ainsi déclarées ne pourront pas intervenir dans ce qu'on appelle des « expressions constantes » (notamment, elles ne pourront pas servir de dimension d'un tableau !).

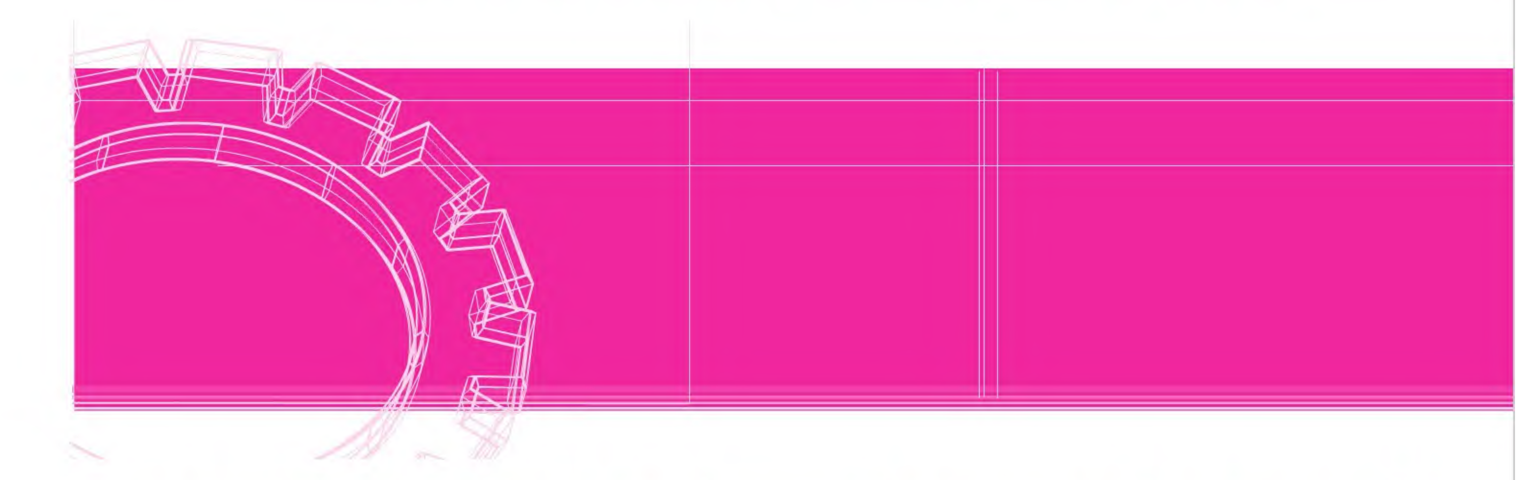
## 6 Autres types introduits par la norme C99

Outre les nouveaux types entiers dont nous avons parlé, la norme C99 introduit :

- le type booléen, sous le nom `bool` ; une variable de ce type ne peut prendre que l'une des deux valeurs : vrai (noté `true`) et faux (noté `false`) ;
- des types complexes, sous les noms `float complex`, `double complex` et `long double complex` ; la constante `I` correspond alors à la constante mathématique `i` (racine de –1).

Chapitre 3

# Les opérateurs et les expressions en langage C



## 1 L'originalité des notions d'opérateur et d'expression en langage C

Le langage C est certainement l'un des langages les plus fournis en opérateurs. Cette richesse se manifeste tout d'abord au niveau des opérateurs classiques (*arithmétiques*, *relationnels*, *logiques*) ou moins classiques (*manipulations de bits*). Mais, de surcroît, le C dispose d'un important éventail d'opérateurs originaux *d'affectation* et *d'incrémentation*.

Ce dernier aspect nécessite une explication. En effet, dans la plupart des langages, on trouve, comme en C :

- d'une part, des *expressions* formées (entre autres) à l'aide d'opérateurs,
- d'autre part, des *instructions* pouvant éventuellement faire intervenir des expressions, comme, par exemple, l'instruction d'affectation :

```
y = a * x +b ;
```

ou encore l'instruction d'affichage :

```
printf ("valeur %d", n + 2*p) ;
```

dans laquelle apparaît l'expression `n + 2 * p`.

Mais, généralement, dans les langages autres que C, l'expression possède une valeur mais ne réalise aucune action, en particulier aucune affectation d'une valeur à une variable. Au contraire, l'affectation y réalise une affectation d'une valeur à une variable mais ne possède pas de valeur. On a affaire à deux notions parfaitement disjointes. En langage C, il en va différemment puisque :

- d'une part, les (nouveaux) opérateurs d'incrémentation pourront non seulement intervenir au sein d'une expression (laquelle, au bout du compte, possédera une valeur), mais également agir sur le contenu de variables. Ainsi, l'expression (car, comme nous le verrons, il s'agit bien d'une expression en C) :

  ```
  ++i
  ```

  réalisera une action, à savoir : augmenter la valeur `i` de `1` ; en même temps, elle aura une valeur, à savoir celle de `i` après incrémentation.

- d'autre part, une affectation apparemment classique telle que :

  ```
  i = 5
  ```

  pourra, à son tour, être considérée comme une expression (ici, de valeur `5`). D'ailleurs, en C, l'affectation (=) est un opérateur. Par exemple, la notation suivante :

  ```
  k = i = 5
  ```

  représente une expression en C (ce n'est pas encore une instruction - nous y reviendrons). Elle sera interprétée comme :

  ```
  k = (i = 5)
  ```

  Autrement dit, elle affectera à `i` la valeur `5` puis elle affectera à `k` la valeur de l'expression `i = 5`, c'est-à-dire `5`.

En fait, en C, les notions d'expression et d'instruction sont étroitement liées puisque **la principale instruction** de ce langage est **une expression terminée par un point-virgule**. On la nomme souvent « instruction expression ». Voici des exemples de telles **instructions** qui reprennent les **expressions** évoquées ci-dessus :

```
++i ;
i = 5 ;
k = i = 5 ;
```

Les deux premières ont l'allure d'une affectation telle qu'on la rencontre classiquement dans la plupart des autres langages. Notez que, dans les deux cas, il y a évaluation d'une expression (`++i` ou `i=5`) dont la valeur est finalement inutilisée. Dans le dernier cas, la valeur de l'expression `i=5`, c'est-à-dire `5`, est à son tour affectée à `k` ; par contre, la valeur finale de l'expression complète est, là encore, inutilisée.

Ce chapitre vous présente la plupart des opérateurs du C ainsi que les règles de priorité et de conversion de type qui interviennent dans les évaluations des expressions. Les (quelques) autres opérateurs concernent essentiellement les pointeurs, l'accès aux tableaux et aux structures et les manipulations de bits. Ils seront exposés dans la suite de cet ouvrage.



# 2 Les opérateurs arithmétiques en C

## 2.1 Présentation des opérateurs

Comme tous les langages, C dispose d'opérateurs classiques « binaires » (c'est-à-dire portant sur deux « opérandes »), à savoir l'addition (`+`), la soustraction (`-`), la multiplication (`*`) et la division (`/`), ainsi que d'un opérateur « unaire » (c'est-à-dire ne portant que sur un seul opérande) correspondant à l'opposé noté `-` (comme dans `-n` ou dans `-x+y`).

Les opérateurs binaires ne sont a priori définis que pour deux opérandes ayant le même type parmi : `int`, `long int`, `float`, `double`, `long double` et ils fournissent un résultat de même type que leurs opérandes.

**Remarque**

En machine, il n'existe, par exemple, que des additions de deux entiers de même taille ou de flottants de même taille. Il n'existe pas d'addition d'un entier et d'un flottant ou de deux flottants de taille différente.

Mais nous verrons, dans le paragraphe 2, que, par le jeu des conversions implicites, le compilateur saura leur donner une signification :

- soit lorsqu'ils porteront sur des opérateurs de type différent,
- soit lorsqu'ils porteront sur des opérandes de type `char` ou `short`.

De plus, il existe un opérateur de modulo noté `%` qui ne peut porter que sur des entiers et qui fournit le reste de la division de son premier opérande par son second. Par exemple, `11%4` vaut `3`, `23%6` vaut `5`.

La norme ANSI ne définit les opérateurs `%` et `/` que pour des valeurs positives de leurs deux opérandes. Dans les autres cas, le résultat dépend de l'implémentation (C99 lève cette ambiguïté).

Notez bien qu'en C le quotient de deux entiers fournit un entier. Ainsi, `5/2` vaut `2` ; en revanche, le quotient de deux flottants (noté, lui aussi, `/`) est bien un flottant (`5.0/2.0` vaut bien approximativement `2.5`).

**Remarque**

Il n'existe pas d'opérateur d'élévation à la puissance. Il est nécessaire de faire appel soit à des produits successifs pour des puissances entières pas trop grandes (par exemple, on calculera $x^3$ comme `x*x*x`), soit à la fonction `power` de la bibliothèque standard (voyez éventuellement l'annexe).

## 2.2 Les priorités relatives des opérateurs

Lorsque plusieurs opérateurs apparaissent dans une même expression, il est nécessaire de savoir dans quel ordre ils sont mis en jeu. En C, comme dans les autres langages, les règles sont naturelles et rejoignent celles de l'algèbre traditionnelle (du moins, en ce qui concerne les opérateurs arithmétiques dont nous parlons ici).

Les opérateurs unaires + et - ont la priorité la plus élevée. On trouve ensuite, à un même niveau, les opérateurs *, / et %. Enfin, sur un dernier niveau, apparaissent les opérateurs binaires + et -.

En cas de priorités identiques, les calculs s'effectuent de gauche à droite. On dit que l'on a affaire à une *associativité de gauche à droite* (nous verrons que quelques opérateurs, autres qu'arithmétiques, utilisent une associativité de droite à gauche).

Enfin, des parenthèses permettent d'outrepasser ces règles de priorité, en forçant le calcul préalable de l'expression qu'elles contiennent. Notez que ces parenthèses peuvent également être employées pour assurer une meilleure lisibilité d'une expression.

Voici quelques exemples dans lesquels l'expression de droite, où ont été introduites des parenthèses superflues, montre dans quel ordre s'effectuent les calculs (les deux expressions proposées conduisent donc aux mêmes résultats) :

```
a + b * c                  a + ( b * c )
a * b + c % d              ( a * b ) + ( c % d )
- c % d                    ( - c ) % d
- a + c % d                ( - a ) + ( c % d )
- a / - b + c              ( ( - a ) / ( - b ) ) + c
- a / - ( b + c )          ( - a ) / ( - ( b + c ) )
```

**Remarques**

Les règles de priorité interviennent pour définir la signification exacte d'une expression. Néanmoins, lorsque deux opérateurs sont théoriquement commutatifs, on ne peut être certain de l'ordre dans lequel ils seront finalement exécutés. Par exemple, une expression telle que a+b+c pourra aussi bien être calculée en ajoutant c à la somme de a et b, qu'en ajoutant a à la somme de b et c. Même l'emploi de parenthèses dans ce cas ne suffit pas à « forcer » l'ordre des calculs. Notez bien qu'une telle remarque n'a d'importance que lorsque l'on cherche à maîtriser parfaitement les erreurs de calcul.

Il est tout à fait possible qu'une opération portant sur deux valeurs entières conduise à un résultat non représentable dans le type concerné, parce que en dehors des limites permises, lesquelles, rappelons-le, dépendent de la machine employée. Dans ce cas, la plupart du temps, on obtient un résultat aberrant : les bits excédentaires sont ignorés, le résultat est analogue à celui obtenu lorsque la somme de deux nombres à 3 chiffres est un nombre à quatre chiffres dont on élimine le chiffre de gauche ; l'exécution du programme se poursuit sans que l'utilisateur ait été informé d'une quelconque anomalie. Notez bien à ce propos qu'un opérateur appliqué par exemple à deux opérandes de type int fournit toujours un résultat de type int, même s'il n'est plus représentable dans ce type et qu'un type long aurait pu convenir.

De la même manière, il se peut qu'à un moment donné vous cherchiez à diviser un entier par zéro. Cette fois, la plupart du temps, cette anomalie est effectivement détectée : un message d'erreur est fourni à l'utilisateur, et l'exécution du programme est interrompue.

Comme les opérations entières, les opérations sur les flottants peuvent conduire à des résultats non représentables dans le type concerné (de valeur absolue trop grande ou trop petite). Dans ce cas, le comportement dépend des environnements de programmation utilisés ; en particulier, il peut y avoir arrêt de l'exécution du programme. Là encore, il faut bien noter qu'un opérateur

appliqué par exemple à deux opérandes de type `float` fournit toujours un résultat de type `float`, même s'il n'est plus représentable dans ce type et qu'un type `double` aurait pu convenir.

En ce qui concerne la division par zéro des flottants, elle conduit toujours à un message et à l'arrêt du programme.

# 3 Les conversions implicites pouvant intervenir dans un calcul d'expression

## 3.1 Notion d'expression mixte

Comme nous l'avons dit, les opérateurs arithmétiques ne sont définis que lorsque leurs deux opérandes sont de même type. Mais vous pouvez écrire ce que l'on nomme des « expressions mixtes » dans lesquelles interviennent des opérandes de types différents. Voici un exemple d'expression autorisée, dans laquelle `n` et `p` sont supposés de type `int`, tandis que `x` est supposé de type `float` :

```
n * x + p
```

Dans ce cas, le compilateur sait, compte tenu des règles de priorité, qu'il doit d'abord effectuer le produit `n*x`. Pour que cela soit possible, il va mettre en place des instructions de conversion de la valeur de `n` dans le type `float` (car on considère que ce type `float` permet de représenter à peu près convenablement une valeur entière, l'inverse étant naturellement faux). Au bout du compte, la multiplication portera sur deux opérandes de type `float` et elle fournira un résultat de type `float`.

Pour l'addition, on se retrouve à nouveau en présence de deux opérandes de types différents (`float` et `int`). Le même mécanisme sera mis en place, et le résultat final sera de type `float`.

**Remarque**

Attention, le compilateur ne peut que prévoir les instructions de conversion (qui seront donc exécutées en même temps que les autres instructions du programme) ; il ne peut pas effectuer lui-même la conversion d'une valeur que généralement il ne peut pas connaître.

## 3.2 Les conversions d'ajustement de type

Une conversion telle que `int -> float` se nomme une « conversion d'ajustement de type ». Une telle conversion ne peut se faire que suivant une hiérarchie qui permet de ne pas dénaturer la valeur initiale (on dit parfois que de telles conversions respectent l'intégrité des données), à savoir :

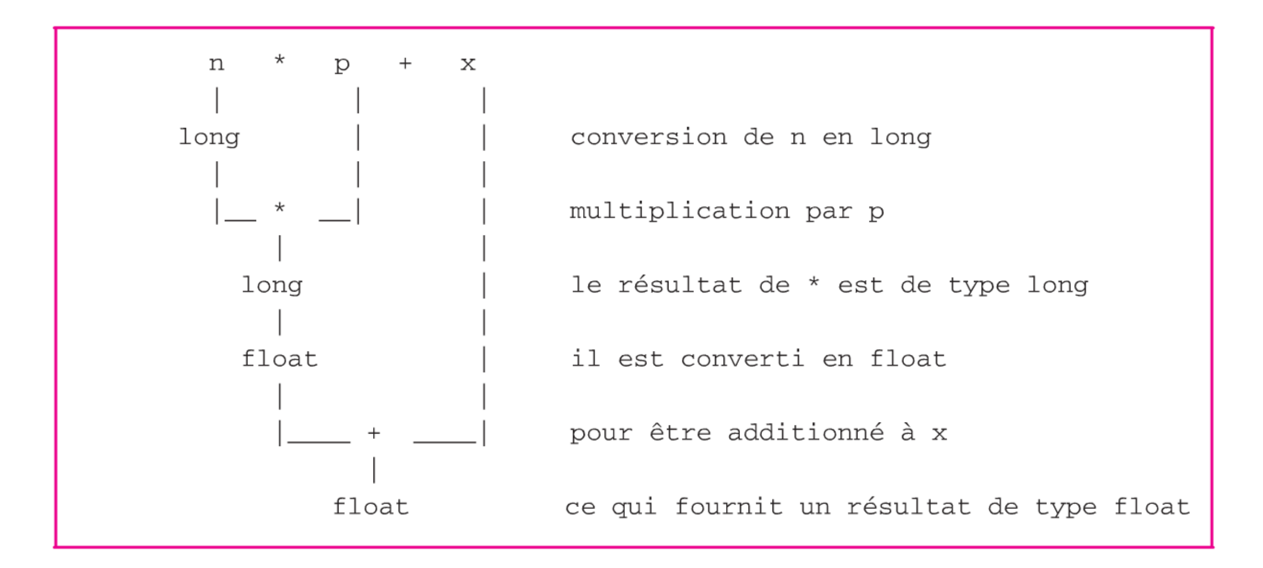
```
int -> long -> float -> double -> long double
```

On peut bien sûr convertir directement un `int` en `double` ; en revanche, on ne pourra pas convertir un `double` en `float` ou en `int`.

Notez que le choix des conversions à mettre en œuvre est effectué en considérant un à un les opérandes concernés et non pas l'expression de façon globale. Par exemple, si `n` est de type `int`, `p` de type `long` et `x` de type `float`, l'expression :

```
n * p + x
```

sera évaluée suivant ce schéma :




## 3.3 Les promotions numériques

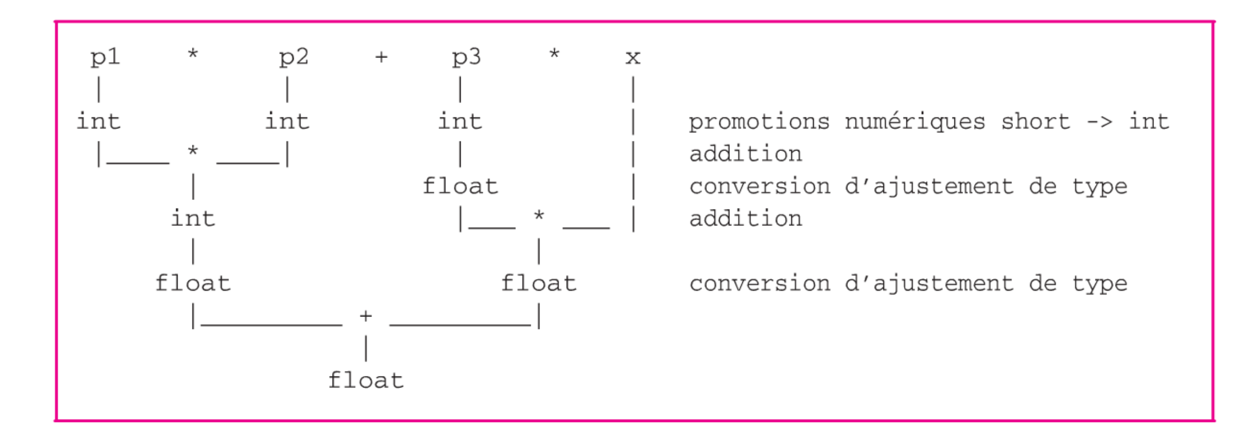
Les conversions d'ajustement de type ne suffisent pas à régler tous les cas. En effet, comme nous l'avons déjà dit, les opérateurs numériques ne sont pas définis pour les types `char` et `short`.

En fait, le langage C prévoit tout simplement que toute valeur de l'un de ces deux types apparaissant dans une expression est d'abord convertie en `int`, et cela sans considérer les types des éventuels autres opérandes. On parle alors, dans ce cas, de « promotions numériques » (ou encore de « conversions systématiques »).

Par exemple, si `p1`, `p2` et `p3` sont de type `short` et `x` de type `float`, l'expression :

```
p1 * p2 + p3 * x
```

est évaluée comme l'indique le schéma ci-après :




Notez bien que les valeurs des trois variables de type `short` sont d'abord soumises à la promotion numérique `short -> int` ; après quoi, on applique les mêmes règles que précédemment.

**Remarque**

En principe, comme nous l'avons déjà dit, les types entiers peuvent être non signés (`unsigned`). Nous y reviendrons dans le chapitre 13. Pour l'instant, sachez que nous vous déconseillons fortement de mélanger, dans une même expression, des types signés et des types non signés, dans la mesure où les conversions qui en résultent sont généralement dénuées de sens (et simplement faites pour préserver un motif binaire).

## 3.4 Le cas du type `char`

A priori, vous pouvez être surpris de l'existence d'une conversion systématique (promotion numérique) de `char` en `int` et vous interroger sur sa signification. En fait, il ne s'agit que d'une question de point de vue. En effet, une valeur de type caractère peut être considérée de deux façons :

- comme le caractère concerné : `a`, `Z`, fin de ligne,
- comme le code de ce caractère, c'est-à-dire un motif de 8 bits ; or à ce dernier on peut toujours faire correspondre un nombre entier (le nombre qui, codé en binaire, fournit le motif en question) ; par exemple, dans le code ASCII, le caractère `E` est représenté par le motif binaire `01000101`, auquel on peut faire correspondre le nombre `65`.

Effectivement, on peut dire qu'en quelque sorte le langage C confond facilement un caractère avec la valeur (entier) du code qui le représente. Notez bien que, comme toutes les machines n'emploient pas le même code pour les caractères, l'entier associé à un caractère donné ne sera pas toujours le même.

Voici quelques exemples d'évaluation d'expressions, dans lesquels on suppose que c1 et c2 sont de type char, tandis que n est de type int.

```
        c1    +    1
         |         |
        int        |            promotion numérique char -> int
         |___ + ___|
              |
             int
```

L'expression c1+1 fournit donc un résultat de type int, correspondant à la valeur du code du caractère contenu dans c1 augmenté d'une unité.

```
        c1    -   c2
         |         |
        int       int           promotions numériques char -> int
         |___ _ ___|
              |
             int
```

Ici, bien que les deux opérandes soient de type char, il y a quand même conversion préalable de leurs valeurs en int (promotions numériques).

```
        c1    +    n
         |         |
        int        |            promotion numérique pour c1
         |___ + ___|
              |
             int
```

**Remarques**

Théoriquement, en plus de ce qui vient d'être dit, il faut tenir compte de l'attribut de signe des caractères. Ainsi, lorsque l'on convertit un unsigned char en int, on obtient toujours un nombre entre 0 et 255, tandis que lorsque l'on convertit un signed char en int, on obtient un nombre compris entre -127 et 128. Nous y reviendrons en détail dans le chapitre 13.

Dans la première version de C (telle qu'elle a été définie initialement par Kernighan et Ritchie, c'est-à-dire avant la normalisation par le comité ANSI), il était prévu une promotion numérique float -> double. Certains compilateurs l'appliquent encore.

Les arguments d'appel d'une fonction peuvent être également soumis à des conversions. Le mécanisme exact est toutefois assez complexe dans ce cas, car il tient compte de la manière

dont la fonction a été déclarée dans le programme qui l'utilise (on peut trouver : aucune déclaration, une déclaration partielle ne mentionnant pas le type des arguments ou une déclaration complète dite prototype mentionnant le type des arguments).

Lorsque le type des arguments n'a pas été déclaré, les valeurs transmises en argument sont soumises aux règles précédentes (donc, en particulier, aux promotions numériques) auxquelles il faut ajouter la promotion numérique `float -> double`. Or, précisément, c'est ainsi que sont traitées les valeurs que vous transmettez à `printf` (ses arguments n'étant pas d'un type connu à l'avance, il est impossible au compilateur d'en connaître le type !). Ainsi :

• tout argument de type `char` ou `short` est converti en `int` ; autrement dit, le code `%c` s'applique aussi à un `int` : il affichera tout simplement le caractère ayant le code correspondant ; de même on obtiendra la valeur numérique du code d'un caractère `c` en écrivant : `printf ("%d", c)`,

• tout argument de type `float` sera converti en `double` (et cela dans toutes les versions du C) ; ainsi le code `%f` pour `printf` correspond-il à un `double`, et il n'est pas besoin de prévoir un code pour un `float`.

# 4 Les opérateurs relationnels

Comme tout langage, C permet de comparer des expressions à l'aide d'opérateurs classiques de comparaison. En voici un exemple :

```
2 * a > b + 5
```

En revanche, C se distingue de la plupart des autres langages sur deux points :

- le résultat de la comparaison est, non pas une valeur booléenne (on dit aussi logique) prenant l'une des deux valeurs *vrai* ou *faux*, mais un entier valant :
  - 0 si le résultat de la comparaison est faux,
  - 1 si le résultat de la comparaison est vrai.

  Ainsi, la comparaison ci-dessus devient en fait une *expression de type entier*. Cela signifie qu'elle pourra éventuellement intervenir dans des calculs arithmétiques ;
- les expressions comparées pourront être d'un type de base quelconque et elles seront soumises aux règles de conversion présentées dans le paragraphe précédent. Cela signifie qu'au bout du compte on ne sera amené à comparer que des expressions de type numérique.

Voici la liste des opérateurs relationnels existant en C. Remarquez bien la notation (==) de l'opérateur d'égalité, le signe = étant, comme nous le verrons, réservé aux affectations. Notez également que = utilisé par mégarde à la place de == ne conduit généralement pas à un diagnostic de compilation, dans la mesure où l'expression ainsi obtenue possède un sens (mais qui n'est pas celui voulu).

*Les opérateurs relationnels*

| OPÉRATEUR | SIGNIFICATION |
|---|---|
| `<` | inférieur à |
| `<=` | inférieur ou égal à |
| `>` | supérieur à |
| `>=` | supérieur ou égal à |
| `==` | égal à |
| `!=` | différent de |

En ce qui concerne leurs priorités, il faut savoir que les quatre premiers opérateurs (`<`, `<=`, `>`, `>=`) sont de même priorité. Les deux derniers (`==` et `!=`) possèdent également la même priorité, mais celle-ci est inférieure à celle des précédents. Ainsi, l'expression :

```
a < b == c < d
```

est interprétée comme :

```
( a < b) == (c < d)
```

ce qui, en C, a effectivement une signification, étant donné que les expressions `a < b` et `c < d` sont, finalement, des quantités entières. En fait, cette expression prendra la valeur `1` lorsque les relations `a < b` et `c < d` auront toutes les deux la même valeur, c'est-à-dire soit lorsqu'elles seront toutes les deux vraies, soit lorsqu'elles seront toutes les deux fausses. Elle prendra la valeur `0` dans le cas contraire.

D'autre part, ces opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques. Cela permet souvent d'éviter certaines parenthèses dans des expressions.

Ainsi :

```
x + y < a + 2
```

est équivalent à :

```
( x + y ) < ( a + 2 )
```

**Remarques**

**Important :** ***comparaisons de caractères.*** Compte tenu des règles de conversion, une comparaison peut porter sur deux caractères. Bien entendu, la comparaison d'égalité ne pose pas de problème particulier ; par exemple (`c1` et `c2` étant de type `char`) :

`c1 == c2` sera vraie si `c1` et `c2` ont la même valeur, c'est-à-dire si `c1` et `c2` contiennent des caractères de même code, donc si `c1` et `c2` contiennent le même caractère,

`c1 == 'e'` sera vraie si le code de `c1` est égal au code de `'e'`, donc si `c1` contient le caractère `e`.

Autrement dit, dans ces circonstances, l'existence d'une conversion `char --> int` n'a guère d'influence. En revanche, pour les comparaisons d'inégalité, quelques précisions s'imposent. En effet, par exemple `c1 < c2` sera vraie si le code du caractère de `c1` a une valeur inférieure au code du caractère de `c2`. Le résultat d'une telle comparaison peut donc varier suivant le codage employé. Cependant, il faut savoir que, quel que soit ce codage :

- l'ordre alphabétique est respecté pour les minuscules d'une part, pour les majuscules d'autre part ; on a toujours `'a' < 'c'`, `'C' < 'S'`...
- les chiffres sont classés par ordre naturel ; on a toujours `'2' < '5'`...

En revanche, aucune hypothèse ne peut être faite sur les places relatives des chiffres, des majuscules et des minuscules, pas plus que sur la place relative des caractères accentués (lorsqu'ils existent) par rapport aux autres caractères !

## 5 Les opérateurs logiques

C dispose de trois opérateurs logiques classiques : **et** (noté `&&`), **ou** (noté `||`) et **non** (noté `!`). Par exemple :

- **`(a<b) && (c<d)`**

  prend la valeur `1` (vrai) si les deux expressions `a<b` et `c<d` sont toutes deux vraies (de valeur `1`), et prend la valeur `0` (faux) dans le cas contraire.

- **`(a<b) || (c<d)`**

  prend la valeur `1` (vrai) si l'une au moins des deux conditions `a<b` et `c<d` est vraie (de valeur `1`), et prend la valeur `0` (faux) dans le cas contraire.

- **`! (a<b)`**

  prend la valeur `1` (vrai) si la condition `a<b` est fausse (de valeur `0`) et prend la valeur `0` (faux) dans le cas contraire. Cette expression est équivalente à : `a>=b`.

Il est important de constater que, ne disposant pas de type logique, C se contente de représenter vrai par `1` et faux par `0`. C'est pourquoi ces opérateurs produisent un résultat numérique (de type `int`).

De plus, on pourrait s'attendre à ce que les opérandes de ces opérateurs ne puissent être que des expressions prenant soit la valeur `0`, soit la valeur `1`. En fait, **ces opérateurs acceptent n'importe quel opérande numérique**, y compris les types flottants, avec les règles de conversion implicite déjà rencontrées. Leur signification reste celle évoquée ci-dessus, à condition de considérer que :

- `0` correspond à faux,
- toute valeur non nulle (et donc pas seulement la valeur `1`) correspond à vrai.

Le tableau suivant récapitule la situation.

*Fonctionnement des opérateurs logiques en C*

| OPERANDE 1 | OPERATEUR | OPERANDE 2 | RESULTAT |
|---|---|---|---|
| 0 | && | 0 | 0 |
| 0 | && | non nul | 0 |
| non nul | && | 0 | 0 |
| non nul | && | non nul | 1 |
| 0 | \|\| | 0 | 0 |
| 0 | \|\| | non nul | 1 |
| non nul | \|\| | 0 | 1 |
| non nul | \|\| | non nul | 1 |
| | ! | 0 | 1 |
| | ! | non nul | 0 |

Ainsi, en C, si `n` et `p` sont des entiers, des expressions telles que :

```
n && p              n || p              !n
```

sont acceptées par le compilateur. Notez que l'on rencontre fréquemment l'écriture :

```
if (!n)
```

plus concise (mais pas forcément plus lisible) que :

```
if ( n == 0 )
```

L'opérateur `!` a une priorité supérieure à celle de tous les opérateurs arithmétiques binaires et aux opérateurs relationnels. Ainsi, pour écrire la condition contraire de :

```
a == b
```

il est nécessaire d'utiliser des parenthèses en écrivant :

```
! ( a == b )
```

En effet, l'expression :

```
! a == b
```

serait interprétée comme :

```
( ! a ) == b
```

L'opérateur || est moins prioritaire que &&. Tous deux sont de priorité inférieure aux opérateurs arithmétiques ou relationnels. Ainsi, les expressions utilisées comme exemples en début de ce paragraphe auraient pu, en fait, être écrites sans parenthèses :

| | | |
|---|---|---|
| `a<b && c<d` | équivaut à | `(a<b) && (c<d)` |
| `a<b \|\| c<d` | équivaut à | `(a<b) \|\| (c<d)` |

Enfin, les deux opérateurs && et || jouissent en C d'une propriété intéressante : **leur second opérande** (celui qui figure à droite de l'opérateur) **n'est évalué que si la connaissance de sa valeur est indispensable** pour décider si l'expression correspondante est vraie ou fausse. Par exemple, dans une expression telle que :

```
a<b && c<d
```

on commence par évaluer `a<b`. Si le résultat est faux (`0`), il est inutile d'évaluer `c<d` puisque, de toute façon, l'expression complète aura la valeur faux (`0`).

La connaissance de cette propriété est indispensable pour maîtriser des « constructions » telles que :

```
if ( i<max && ( (c=getchar()) != '\n' ) )
```

En effet, le second opérande de l'opérateur &&, à savoir :

```
c = getchar() != '\n'
```

fait appel à la lecture d'un caractère au clavier. Celle-ci n'aura donc lieu que si la première condition (`i<max`) est vraie.

## 6 L'opérateur d'affectation ordinaire

Nous avons déjà eu l'occasion de remarquer que :

```
i = 5
```

était une expression qui :

- réalisait une action : l'affectation de la valeur `5` à `i`,
- possédait une valeur : celle de `i` après affectation, c'est-à-dire `5`.

Cet opérateur d'affectation (=) peut faire intervenir d'autres expressions comme dans :

```
c = b + 3
```

La faible priorité de cet opérateur = (elle est inférieure à celle de tous les opérateurs arithmétiques et de comparaison) fait qu'il y a d'abord évaluation de l'expression `b + 3`. La valeur ainsi obtenue est ensuite affectée à `c`.

En revanche, il n'est pas possible de faire apparaître une expression comme premier opérande de cet opérateur =. Ainsi, l'expression suivante n'aurait pas de sens :

```
c + 5 = x
```

## 6.1 Notion de lvalue

Nous voyons donc que cet opérateur d'affectation impose des restrictions sur son premier opérande. En effet, ce dernier doit être une référence à un emplacement mémoire dont on pourra effectivement modifier la valeur.

Dans les autres langages, on désigne souvent une telle référence par le nom de « variable » ; on précise généralement que ce terme recouvre par exemple les éléments de tableaux ou les composantes d'une structure. En langage C, cependant, la syntaxe du langage est telle que cette notion de variable n'est pas assez précise. Il faut introduire un mot nouveau : la **lvalue**. Ce terme désigne une « valeur à gauche », c'est-à-dire tout ce qui peut apparaître à gauche d'un opérateur d'affectation.

Certes, pour l'instant, vous pouvez trouver que dire qu'à gauche d'un opérateur d'affectation doit apparaître une *lvalue* n'apporte aucune information. En fait, d'une part, nous verrons qu'en C d'autres opérateurs que = font intervenir une *lvalue* ; d'autre part, au fur et à mesure que nous rencontrerons de nouveaux types d'objets, nous préciserons s'ils peuvent être ou non utilisés comme *lvalue*.

Pour l'instant, les seules *lvalue* que nous connaissons restent les variables de n'importe quel type de base déjà rencontré.

## 6.2 L'opérateur d'affectation possède une associativité de droite à gauche

Contrairement à tous ceux que nous avons rencontrés jusqu'ici, cet opérateur d'affectation possède une associativité de *droite à gauche*. C'est ce qui permet à une expression telle que :

```
i = j = 5
```

d'évaluer d'abord l'expression `j = 5` avant d'en affecter la valeur (`5`) à la variable `j`. Bien entendu, la valeur finale de cette expression est celle de i après affectation, c'est-à-dire `5`.

## 6.3 L'affectation peut entraîner une conversion

Là encore, la grande liberté offerte par le langage C en matière de mixage de types se traduit par la possibilité de fournir à cet opérateur d'affectation des opérandes de types différents.

Cette fois, cependant, contrairement à ce qui se produisait pour les opérateurs rencontrés précédemment et qui mettaient en jeu des conversions implicites, il n'est plus question, ici, d'effectuer une quelconque conversion de la *lvalue* qui apparaît à gauche de cet opérateur.

Une telle conversion reviendrait à changer le type de la *lvalue* figurant à gauche de cet opérateur, ce qui n'a pas de sens.

En fait, lorsque le type de l'expression figurant à droite n'est pas du même type que la *lvalue* figurant à gauche, il y a **conversion systématique** de la valeur de l'expression (qui est évaluée suivant les règles habituelles) dans le type de la *lvalue*. Une telle conversion **imposée** ne respecte plus nécessairement la hiérarchie des types qui est de rigueur dans le cas des conversions implicites. Elle peut donc conduire, suivant les cas, à une dégradation plus ou moins importante de l'information (par exemple lorsque l'on convertit un `double` en `int`, on perd la partie décimale du nombre).

Nous ferons le point sur ces différentes possibilités de conversions imposées par les affectations dans le paragraphe 9.

# 7 Les opérateurs d'incrémentation et de décrémentation

## 7.1 Leur rôle

Dans des programmes écrits dans un langage autre que C, on rencontre souvent des expressions (ou des instructions) telles que :

```
i = i + 1
n = n - 1
```

qui incrémentent ou qui décrémentent de `1` la valeur d'une variable (ou plus généralement d'une *lvalue*).

En C, ces actions peuvent être réalisées par des opérateurs « unaires » portant sur cette *lvalue*. Ainsi, l'expression :

```
++i
```

a pour effet d'incrémenter de `1` la valeur de `i`, et sa valeur est celle de `i` **après incrémentation**.

Là encore, comme pour l'affectation, nous avons affaire à une expression qui non seulement possède une valeur, mais qui, de surcroît, réalise une action (incrémentation de `i`).

Il est important de voir que la valeur de cette expression est celle de `i` après incrémentation. Ainsi, si la valeur de `i` est `5`, l'expression :

```
n = ++i - 5
```

affectera à `i` la valeur `6` et à `n` la valeur `1`.

En revanche, lorsque cet opérateur est placé *après* la *lvalue* sur laquelle il porte, la valeur de l'expression correspondante est celle de la variable **avant incrémentation**.

Ainsi, si `i` vaut `5`, l'expression :

```
n = i++ - 5
```

affectera à `i` la valeur `6` et à `n` la valeur `0` (car ici la valeur de l'expression `i++` est `5`).

On dit que `++` est :

- un opérateur de **préincrémentation** lorsqu'il est placé à gauche de la *lvalue* sur laquelle il porte,
- un opérateur de **postincrémentation** lorsqu'il est placé à droite de la *lvalue* sur laquelle il porte.

Bien entendu, lorsque seul importe l'effet d'incrémentation d'une *lvalue*, cet opérateur peut être indifféremment placé avant ou après. Ainsi, ces deux instructions (ici, il s'agit bien d'instructions car les expressions sont terminées par un point-virgule - leur valeur se trouve donc inutilisée) sont équivalentes :

```
i++ ;
++i ;
```

De la même manière, il existe un opérateur de décrémentation noté `--` qui, suivant les cas, sera :

- un opérateur de **prédécrémentation** lorsqu'il est placé à gauche de la *lvalue* sur laquelle il porte,
- un opérateur de **postdécrémentation** lorsqu'il est placé à droite de la *lvalue* sur laquelle il porte.

## 7.2 Leurs priorités

Les priorités élevées de ces opérateurs unaires (voir tableau en fin de chapitre) permettent d'écrire des expressions assez compliquées sans qu'il soit nécessaire d'employer des parenthèses pour isoler la *lvalue* sur laquelle ils portent. Ainsi, l'expression suivante a un sens :

```
3 * i++ * j-- + k++
```

(si * avait été plus prioritaire que la postincrémentation, ce dernier aurait été appliqué à l'expression `3*i` qui n'est pas une *lvalue* ; l'expression n'aurait alors pas eu de sens).

**Remarque**

Il est toujours possible (mais non obligatoire) de placer un ou plusieurs espaces entre un opérateur et les opérandes sur lesquels il porte. Nous utilisons souvent cette latitude pour accroître la lisibilité de nos instructions. Cependant, dans le cas des opérateurs d'incrémentation, nous avons plutôt tendance à ne pas le faire, cela pour mieux rapprocher l'opérateur de la *lvalue* sur laquelle il porte.

### 7.3 Leur intérêt

Ces opérateurs allègent l'écriture de certaines expressions et offrent surtout le grand avantage d'éviter la redondance qui est de mise dans la plupart des autres langages. En effet, dans une notation telle que :

```
i++
```

on ne cite qu'une seule fois la *lvalue* concernée alors qu'on est amené à le faire deux fois dans la notation :

```
i = i + 1
```

Les risques d'erreurs de programmation s'en trouvent ainsi quelque peu limités. Bien entendu, cet aspect prendra d'autant plus d'importance que la *lvalue* correspondante sera d'autant plus complexe.

D'une manière générale, nous utiliserons fréquemment ces opérateurs dans la manipulation de tableaux ou de chaînes de caractères. Ainsi, anticipant sur les chapitres suivants, nous pouvons indiquer qu'il sera possible de lire l'ensemble des valeurs d'un tableau nommé `t` en répétant la seule instruction :

```
t [i++] = getchar() ;
```

Celle-ci réalisera à la fois :

- la lecture d'un caractère au clavier,
- l'affectation de ce caractère à l'élément de rang `i` du tableau `t`,
- l'incrémentation de `1` de la valeur de `i` (qui sera ainsi préparée pour la lecture du prochain élément).

## 8 Les opérateurs d'affectation élargie

Nous venons de voir comment les opérateurs d'incrémentation permettaient de simplifier l'écriture de certaines affectations. Par exemple :

```
i++
```

remplaçait avantageusement :

```
i = i + 1
```

Mais C dispose d'opérateurs encore plus puissants. Ainsi, vous pourrez remplacer :

```
i = i + k
```

par :

```
i += k
```

ou, mieux encore :

```
a = a * b
```

par :

```
a *= b
```

D'une manière générale, C permet de condenser les affectations de la forme :

```
lvalue   =   lvalue   opérateur   expression
```

en :

```
lvalue   opérateur=   expression
```

Cette possibilité concerne tous les opérateurs binaires arithmétiques et de manipulation de bits. Voici la liste complète de tous ces nouveaux opérateurs nommés « opérateurs d'affectation élargie » :

```
+=   -=   *=   /=   %=   |=   ^=   &=   <<=   >>=
```

**Remarque**

Les cinq derniers correspondent en fait à des « opérateurs de manipulation de bits » (`|`, `^`, `&`, `<<` et `>>`) que nous n'aborderons que dans le chapitre 13.

Ces opérateurs, comme ceux d'incrémentation, permettent de condenser l'écriture de certaines instructions et contribuent à éviter la redondance introduite fréquemment par l'opérateur d'affectation classique.

**Remarque**

Ne confondez pas l'opérateur de comparaison `<=` avec un opérateur d'affectation élargie. Notez bien que les opérateurs de comparaison ne sont pas concernés par cette possibilité.

## 9 Les conversions forcées par une affectation

Nous avons déjà vu comment le compilateur peut être amené à introduire des conversions implicites dans l'évaluation des expressions. Dans ce cas, il applique les règles de promotions numériques et d'ajustement de type.

Par ailleurs, une affectation introduit une conversion d'office dans le type de la *lvalue* réceptrice, dès lors que cette dernière est d'un type différent de celui de l'expression correspondante. Par exemple, si `n` est de type `int` et `x` de type `float`, l'affectation :

```
n = x + 5.3 ;
```

entraînera tout d'abord l'évaluation de l'expression située à droite, ce qui fournira une valeur de type `float` ; cette dernière sera ensuite convertie en `int` pour pouvoir être affectée à `n`.

D'une manière générale, lors d'une affectation, toutes les conversions (d'un type numérique vers un autre type numérique) sont acceptées par le compilateur mais le résultat en est plus ou moins satisfaisant. En effet, si aucun problème ne se pose (autre qu'une éventuelle perte de précision) dans le cas de conversion ayant lieu suivant le bon sens de la hiérarchie des types, il n'en va plus de même dans les autres cas.

Par exemple, la conversion `float -> int` (telle que celle qui est mise en jeu dans l'instruction précédente) ne fournira un résultat acceptable que si la partie entière de la valeur flottante est représentable dans le type `int`. Si une telle condition n'est pas réalisée, non seulement le résultat obtenu pourra être différent d'un environnement à un autre mais, de surcroît, on pourra aboutir, dans certains cas, à une erreur d'exécution.

De la même manière, la conversion d'un `int` en `char` sera satisfaisante si la valeur de l'entier correspond à un code d'un caractère.

Sachez, toutefois, que les conversions d'un type entier vers un autre type entier ne conduisent, au pis, qu'à une valeur inattendue mais jamais à une erreur d'exécution.

## 10 L'opérateur de cast

S'il le souhaite, le programmeur peut forcer la conversion d'une expression quelconque dans un type de son choix, à l'aide d'un opérateur un peu particulier nommé en anglais « **cast** ».

Si, par exemple, `n` et `p` sont des variables entières, l'expression :

```
(double) ( n/p )
```

aura comme valeur celle de l'expression entière `n/p` convertie en `double`.

La notation `(double)` correspond en fait à un opérateur unaire dont le rôle est d'effectuer la conversion dans le type `double` de l'expression sur laquelle il porte. Notez bien que cet opérateur force la conversion du *résultat* de l'expression et non celle des différentes valeurs qui concourent à son évaluation. Autrement dit, ici, il y a d'abord calcul, dans le type `int`, du quotient de `n` par `p` ; c'est seulement ensuite que le résultat sera converti en `double`. Si `n` vaut `10` et que `p` vaut `3`, cette expression aura comme valeur `3`.

D'une manière générale, il existe autant d'opérateurs de « cast » que de types différents (y compris les types dérivés comme les pointeurs que nous rencontrerons ultérieurement). Leur priorité élevée (voir tableau en fin de chapitre) fait qu'il est généralement nécessaire de placer entre parenthèses l'expression concernée. Ainsi, l'expression :

```
(double) n/p
```

conduirait d'abord à convertir n en double ; les règles de conversions implicites amèneraient alors à convertir p en double avant qu'ait lieu la division (en double). Le résultat serait alors différent de celui obtenu par l'expression proposée en début de ce paragraphe (avec les mêmes valeurs de n et de p, on obtiendrait une valeur de l'ordre de 3.33333...).

Bien entendu, comme pour les conversions forcées par une affectation, toutes les conversions numériques sont réalisables par un opérateur de « cast », mais le résultat en est plus ou moins satisfaisant (revoyez éventuellement le paragraphe précédent).

## 11 L'opérateur conditionnel

Considérons l'instruction suivante :

```
if ( a>b )
        max = a ;
   else
        max = b ;
```

Elle attribue à la variable max la plus grande des deux valeurs de a et de b. La valeur de max pourrait être définie par cette phrase :

```
Si a>b alors a sinon b
```

En langage C, il est possible, grâce à l'aide de **l'opérateur conditionnel**, de traduire presque littéralement la phrase ci-dessus de la manière suivante :

```
max = a>b ? a : b
```

L'expression figurant à droite de l'opérateur d'affectation est en fait constituée de trois expressions (a>b, a et b) qui sont les trois opérandes de l'opérateur conditionnel, lequel se matérialise par *deux symboles séparés :* ? et :.

D'une manière générale, cet opérateur évalue la première expression qui joue le rôle d'une condition. Comme toujours en C, celle-ci peut être en fait de n'importe quel type. Si sa valeur est différente de zéro, il y a évaluation du second opérande, ce qui fournit le résultat ; si sa valeur est nulle, en revanche, il y a évaluation du troisième opérande, ce qui fournit le résultat.

Voici un autre exemple d'une expression calculant la valeur absolue de 3*a + 1 :

```
3*a+1 > 0  ?  3*a+1 :  -3*a-1
```

L'opérateur conditionnel dispose d'une faible priorité (il arrive juste avant l'affectation), de sorte qu'il est rarement nécessaire d'employer des parenthèses pour en délimiter les différents opérandes (bien que cela puisse parfois améliorer la lisibilité du programme). Voici, toutefois, un cas où les parenthèses sont indispensables :

```
z = (x=y) ? a : b
```

Le calcul de cette expression amène tout d'abord à affecter la valeur de `y` à `x`. Puis, si cette valeur est non nulle, on affecte la valeur de `a` à `z`. Si, au contraire, cette valeur est nulle, on affecte la valeur de `b` à `z`.

Il est clair que cette expression est différente de :

```
z = x = y ? a : b
```

laquelle serait évaluée comme :

```
z = x = ( y ? a : b )
```

Bien entendu, une expression conditionnelle peut, comme toute expression, apparaître à son tour dans une expression plus complexe. Voici, par exemple, une instruction (notez qu'il s'agit effectivement d'une instruction, car elle se termine par un point-virgule) affectant à `z` la plus grande des valeurs de `a` et de `b` :

```
z = ( a>b ? a : b ) ;
```

De même, rien n'empêche que l'expression conditionnelle soit évaluée sans que sa valeur soit utilisée comme dans cette instruction :

```
a>b ? i++ : i-- ;
```

Ici, suivant que la condition `a>b` est vraie ou fausse, on incrémentera ou on décrémentera la variable `i`.

## 12 L'opérateur séquentiel

Nous avons déjà vu qu'en C la notion d'expression était beaucoup plus générale que dans la plupart des autres langages. L'opérateur dit « séquentiel » va élargir encore un peu plus cette notion d'expression. En effet, celui-ci permet, en quelque sorte, d'exprimer *plusieurs calculs successifs au sein d'une même expression.* Par exemple :

```
a * b , i + j
```

est une expression qui évalue d'abord `a*b`, puis `i+j` et qui prend comme valeur la dernière calculée (donc ici celle de `i+j`). Certes, dans ce cas d'école, le calcul préalable de `a*b` est inutile puisqu'il n'intervient pas dans la valeur de l'expression globale et qu'il ne réalise aucune action.

En revanche, une expression telle que :

```
i++, a + b
```

peut présenter un intérêt puisque la première expression (dont la valeur ne sera pas utilisée) réalise en fait une incrémentation de la variable `i`.

Il en est de même de l'expression suivante :

```
i++, j = i + k
```

dans laquelle, il y a :

- évaluation de l'expression `i++`,
- évaluation de l'affectation `j = i + k`. Notez qu'alors on utilise la valeur de `i` après incrémentation par l'expression précédente.

Cet opérateur séquentiel, qui dispose d'une associativité de gauche à droite, peut facilement faire intervenir plusieurs expressions (sa faible priorité évite l'usage de parenthèses) :

```
i++, j = i+k, j--
```

Certes, un tel opérateur peut être utilisé pour réunir plusieurs instructions en une seule. Ainsi, par exemple, ces deux formulations sont équivalentes :

```
i++, j = i+k, j-- ;
i++ ; j = i+k ; j-- ;
```

Dans la pratique, ce n'est cependant pas là le principal usage que l'on fera de cet opérateur séquentiel. En revanche, ce dernier pourra fréquemment intervenir dans les instructions de choix ou dans les boucles ; là où celles-ci s'attendent à trouver une seule expression, l'opérateur séquentiel permettra d'en placer plusieurs, et donc d'y réaliser plusieurs calculs ou plusieurs actions. En voici deux exemples :

```
if (i++, k>0) ......
```

remplace :

```
i++ ; if (k>0) ......
```

et :

```
for (i=1, k=0 ; ... ; ... ) .......
```

remplace :

```
i=1 ; for (k=0 ; ... ; ... ) ......
```

Compte tenu de ce que l'appel d'une fonction n'est en fait rien d'autre qu'une expression, la construction suivante est parfaitement valide en C :

```
for (i=1, k=0, printf("on commence") ; ... ; ...) ......
```

Nous verrons même que, dans le cas des boucles conditionnelles, cet opérateur permet de réaliser des constructions ne possédant pas d'équivalent simple.

# 13 L'opérateur sizeof

L'opérateur `sizeof`, dont l'emploi ressemble à celui d'une fonction, fournit la taille en octets (n'oubliez pas que l'octet est, en fait, la plus petite partie adressable de la mémoire). Par exemple, dans une implémentation où le type `int` est représenté sur 2 octets et le type `double` sur 8 octets, si l'on suppose que l'on a affaire à ces déclarations :

```
int n ;
double z ;
```

- l'expression `sizeof(n)` vaudra `2`,
- l'expression `sizeof(z)` vaudra `8`.

Cet opérateur peut également s'appliquer à un type de nom donné. Ainsi, dans l'implémentation précédemment citée :

- `sizeof(int)` vaudra `2`,
- `sizeof(double)` vaudra `8`.

Quelle que soit l'implémentation, `sizeof(char)` vaudra toujours `1` (par définition, en quelque sorte).

Cet opérateur offre un intérêt :

- lorsque l'on souhaite écrire des programmes portables dans lesquels il est nécessaire de connaître la taille exacte de certains objets,
- pour éviter d'avoir à calculer soi-même la taille d'objets d'un type relativement complexe pour lequel on n'est pas certain de la manière dont il sera implémenté par le compilateur. Ce sera notamment le cas des structures.

# 14 Récapitulatif des priorités de tous les opérateurs

Le tableau ci-après fournit la liste **complète** des opérateurs du langage C, classés par ordre de priorité décroissante, accompagnés de leur mode d'associativité.

*Les opérateurs du langage C et leurs priorités*

| CATEGORIE | OPERATEURS | ASSOCIATIVITE |
|---|---|---|
| référence | () [] -> . | ---> |
| unaire | + - ++ -- ! ~ * & (cast) sizeof | <--- |
| arithmétique | * / % | ---> |
| arithmétique | + - | ---> |
| décalage | << >> | ---> |
| relationnels | < <= > >= | ---> |
| relationnels | == != | ---> |
| manip. de bits | & | ---> |
| manip. de bits | ^ | ---> |
| manip de bits | | | ---> |
| logique | && | ---> |
| logique | || | ---> |
| conditionnel | ? : | ---> |
| affectation | = += -= *= /= %= &= ^= |= <<= >>= | <--- |
| séquentiel | , | ---> |

En langage C, un certain nombre de notations servant à référencer des objets sont considérées comme des opérateurs et, en tant que tels, soumises à des règles de priorité. Ce sont essentiellement :

- les références à des éléments d'un tableau réalisées par `[]`
- des références à des champs d'une structure : opérateurs `->` et `,`
- des opérateurs d'adressage : `*` et `&`

Ces opérateurs seront étudiés ultérieurement dans les chapitres correspondant aux tableaux, structures et pointeurs. Néanmoins, ils figurent dans le tableau proposé. De même, vous y trouverez les opérateurs de manipulation de bits dont nous ne parlerons que dans le chapitre 13.

# Exercices

Tous ces exercices sont corrigés en fin de volume.

**1)** Soit les déclarations suivantes :

```
int n = 10 , p = 4 ;
long q = 2 ;
float x = 1.75 ;
```

Donner le type et la valeur de chacune des expressions suivantes :

a) `n + q`
b) `n + x`
c) `n % p +q`
d) `n < p`
e) `n >= p`
f) `n > q`
g) `q + 3 * (n > p)`
h) `q && n`
i) `(q-2) && (n-10)`
j) `x * (q==2)`
k) `x *(q=5)`

**2)** Écrire plus simplement l'instruction suivante :

```
z = (a>b ? a : b) + (a <= b ? a : b) ;
```

**3)** `n` étant de type `int`, écrire une expression qui prend la valeur :

`-1` si `n` est négatif,

`0` si `n` est nul,

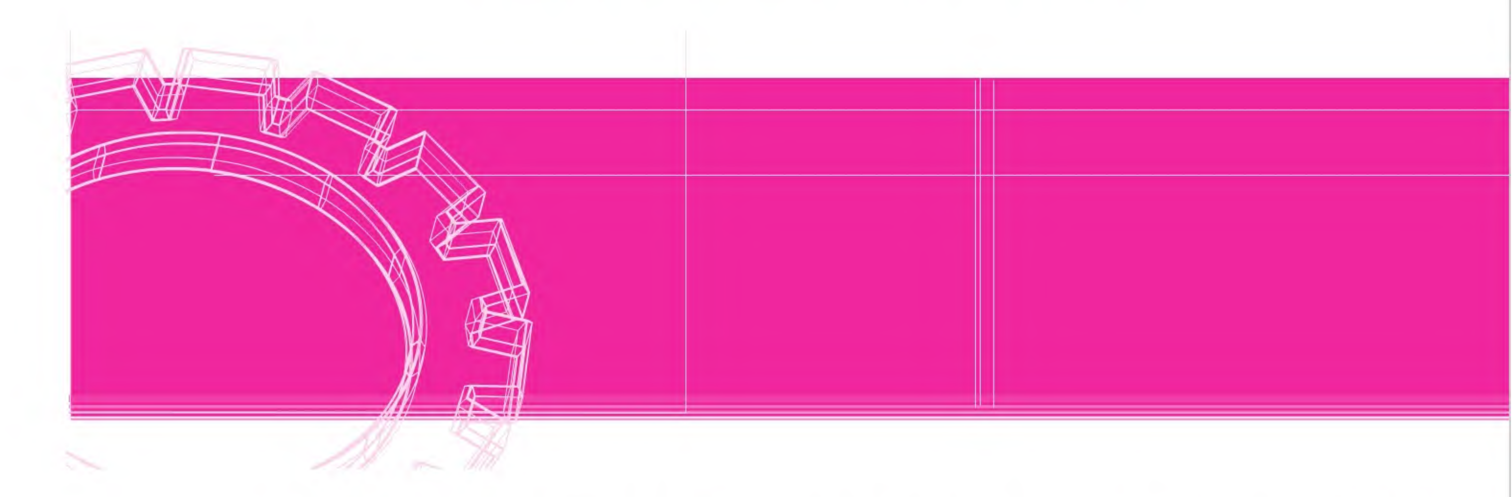
`1` si `n` est positif.

**4)** Quels résultats fournit le programme suivant ?

```
#include <stdio.h>
main()
{
   int n=10, p=5, q=10, r ;
   r = n == (p = q) ;
   printf ("A : n = %d  p = %d  q = %d  r = %d\n", n, p, q, r) ;
   n = p = q = 5 ;
   n += p += q ;
   printf ("B : n = %d  p = %d  q = %d\n", n, p, q) ;
   q = n < p ? n++ : p++ ;
   printf ("C : n = %d  p = %d  q = %d\n", n, p, q) ;
   q = n > p ? n++ : p++ ;
   printf ("D : n = %d  p = %d  q = %d\n", n, p, q) ;
}
```

# Chapitre 4

# Les entrées-sorties conversationnelles



Jusqu'ici, nous avons utilisé de façon intuitive les fonctions `printf` et `scanf` pour afficher des informations à l'écran ou pour en lire au clavier. Nous vous proposons maintenant d'étudier en détail les différentes possibilités de ces fonctions, ce qui nous permettra de répondre à des questions telles que :

- quelles sont les écritures autorisées pour des nombres fournis en données ? Que se passe-t-il lorsque l'utilisateur ne les respecte pas ?
- comment organiser les données lorsque l'on mélange les types numériques et les types caractères ?
- que se produit-il lorsque, en réponse à `scanf`, on fournit trop ou trop peu d'informations ?
- comment agir sur la présentation des informations à l'écran ?

Nous nous limiterons ici à ce que nous avons appelé les « entrées-sorties conversationnelles ». Plus tard, nous verrons que ces mêmes fonctions (moyennant la présence d'un argument supplémentaire) permettent également d'échanger des informations avec des fichiers.

En ce qui concerne la lecture au clavier, nous serons amené à mettre en évidence certaines lacunes de `scanf` en matière de comportement lors de réponses incorrectes et à vous fournir quelques idées sur la manière d'y remédier.

# 1 Les possibilités de la fonction printf

Nous avons déjà vu que le premier argument de printf est une chaîne de caractères qui spécifie à la fois :

- des caractères à afficher tels quels,
- des codes de format repérés par %. Un code de conversion (tel que c, d ou f) y précise le type de l'information à afficher.

D'une manière générale, il existe d'autres caractères de conversion soit pour d'autres types de valeurs, soit pour agir sur la précision de l'information que l'on affiche. De plus, un code de format peut contenir des informations complémentaires agissant sur le cadrage, le gabarit ou la précision. Ici, nous nous limiterons aux possibilités les plus usitées de printf . Nous avons toutefois mentionné le code de conversion relatif aux chaînes (qui ne seront abordées que dans le chapitre 8) et les entiers non signés (chapitre 13). Sachez cependant que le paragraphe 1.2 de l'annexe vous en fournit un panorama complet.

## 1.1 Les principaux codes de conversion

**c** char : caractère affiché « en clair » (convient aussi à short ou à int compte tenu des conversions systématiques)

**d** int (convient aussi à char ou à int, compte tenu des conversions systématiques)

**u** unsigned int (convient aussi à unsigned char ou à unsigned short, compte tenu des conversions systématiques)

**ld** long

**lu** unsigned long

**f** double ou float (compte tenu des conversions systématiques float -> double) écrit en notation décimale avec six chiffres après le point (par exemple : 1.234500 ou 123.456789)

**e** double ou float (compte tenu des conversions systématiques float -> double) écrit en notation exponentielle (mantisse entre 1 inclus et 10 exclu) avec six chiffres après le point décimal, sous la forme x.xxxxxxe+yyy ou x.xxxxxx-yyy pour les nombres positifs et -x.xxxxxxe+yyy ou -x.xxxxxxe-yyy pour les nombres négatifs

**s** chaîne de caractères dont on fournit l'adresse (notion qui sera étudiée ultérieurement)

## 1.2 Action sur le gabarit d'affichage

Par défaut, les entiers sont affichés avec le nombre de caractères nécessaires (sans espaces avant ou après). Les flottants sont affichés avec six chiffres après le point (aussi bien pour le code e que f).

Un nombre placé après % dans le code de format précise un gabarit d'affichage, c'est-à-dire un nombre **minimal** de caractères à utiliser. Si le nombre peut s'écrire avec moins de caractères, `printf` le fera précéder d'un nombre suffisant d'espaces ; en revanche, si le nombre ne peut s'afficher convenablement dans le gabarit imparti, `printf` utilisera le nombre de caractères nécessaires.

Voici quelques exemples, dans lesquels nous fournissons, à la suite d'une instruction `printf`, à la fois des valeurs possibles des expressions à afficher et le résultat obtenu à l'écran. Notez que le symbole ^ représente un espace.

```
printf ("%3d", n) ;      /* entier avec 3 caractères minimum */
      n = 20                  ^20
      n = 3                   ^^3
      n = 2358                2358
      n = -5200               -5200

printf ("%f", x) ;       /* notation décimale gabarit par défaut  */
                         /*   (6 chiffres après point)            */
      x = 1.2345              1.234500
      x = 12.3456789          12.345679

printf ("%10f", x) ;     /* notation décimale - gabarit mini 10 */
                         /*   (toujours 6 chiffres après point) */
      x = 1.2345              ^^1.234500
      x = 12.345              ^12.345000
      x = 1.2345E5            123450.000000

printf ("%e", x) ;   /* notation exponentielle - gabarit par défaut */
                     /*      (6 chiffres après point)               */
      x = 1.2345              1.234500e+000
      x = 123.45              1.234500e+002
      x = 123.456789E8        1.234568e+010
      x = -123.456789E8       -1.234568e+010
```

## 1.3 Actions sur la précision

Pour les types flottants, on peut spécifier un nombre de chiffres (éventuellement inférieur à 6) après le point décimal (aussi bien pour la notation décimale que pour la notation exponentielle). Ce nombre doit apparaître, précédé d'un point, avant le code de format (et éventuellement après le gabarit).

Voici quelques exemples :

```
printf ("%10.3f", x) ;  /* notation décimale, gabarit mini 10 */
                        /*    et 3 chiffres après point       */
      x = 1.2345                ^^^^^1.235
      x = 1.2345E3              ^^1234.500
      x = 1.2345E7              12345000.000
printf ("%12.4e", x) ;  /* notation exponentielle, gabarit mini 12*/
                        /*    et 4 chiffres après point          */
      x = 1.2345                ^1.2345e+000
      x = 123.456789E8          ^1.2346e+010
```

**Remarques**

Le signe moins (-), placé immédiatement après le symbole % (comme dans %-4d ou %-10.3f), demande de cadrer l'affichage à gauche au lieu de le cadrer (par défaut) à droite ; les éventuels espaces supplémentaires sont donc placés à droite et non plus à gauche de l'information affichée.

Le caractère * figurant à la place d'un gabarit ou d'une précision signifie que la valeur effective est fournie dans la liste des arguments de printf. En voici un exemple dans lequel nous appliquons ce mécanisme à la précision :

```
printf ("%8.*f", n, x) ;
n = 1    x = 1.2345                ^^^^1.2
n = 3    x = 1.2345                ^^^1.234
```

La fonction printf fournit en fait une valeur de retour. Il s'agit du nombre de caractères qu'elle a réellement affichés (ou la valeur -1 en cas d'erreur). Par exemple, avec l'instruction suivante, on s'assure que l'opération d'affichage s'est bien déroulée :

```
if (printf ("....", ...) != -1 ) .....
```

De même, on obtient le nombre de caractères effectivement affichés par :

```
n = printf ("....", ....)
```

## 1.4 La syntaxe de printf

D'une manière générale, nous pouvons dire que l'appel à printf se présente ainsi :

*La fonction* printf

```
printf  ( format,  liste_d'expressions )
```

- `format` :
  - constante chaîne (entre " "),
  - pointeur sur une chaîne de caractères (cette notion sera étudiée ultérieurement).
- `liste_d'expressions` : suite d'expressions séparées par des virgules d'un type en accord avec le code format correspondant.

**Remarque**

Nous verrons que les deux notions de constante chaîne et de pointeur sur une chaîne sont identiques.

## 1.5 En cas d'erreur de programmation

Deux types d'erreur de programmation peuvent apparaître dans l'emploi de `printf`.

### *a) code de format en désaccord avec le type de l'expression*

Lorsque le code de format, bien qu'erroné, correspond à une information de **même taille** (c'est-à-dire occupant la même place en mémoire) que celle relative au type de l'expression, les conséquences de l'erreur se limitent à une *mauvaise interprétation de l'expression*. C'est ce qui se passe, par exemple, lorsque l'on écrit une valeur de type `int` en `%u` ou une valeur de type `unsigned int` en `%d`.

En revanche, lorsque le code format correspond à une information de **taille différente** de celle relative au type de l'expression, les conséquences sont généralement plus désastreuses, du moins si d'autres valeurs doivent être affichées à la suite. En effet, tout se passe alors comme si, dans la suite d'octets (correspondant aux différentes valeurs à afficher) reçue par `printf`, le repérage des emplacements des valeurs suivantes se trouvait soumis à un décalage.

### *b) nombre de codes de format différent du nombre d'expressions de la liste*

Dans ce cas, il faut savoir que **C cherche toujours à satisfaire le contenu du format**.

Ce qui signifie que, si des expressions de la liste n'ont pas de code format, elles ne seront pas affichées. C'est le cas dans cette instruction où la valeur de `p` ne sera pas affichée :

```
printf ("%d", n, p) ;
```

En revanche, si vous prévoyez trop de codes de format, les conséquences seront là encore assez désastreuses puisque `printf` cherchera à afficher n'importe quoi. C'est le cas dans cette instruction où deux valeurs seront affichées, la seconde étant (relativement) aléatoire :

```
printf ("%d %d ", n) ;
```

### 1.6 La macro putchar

L'expression :

```
putchar (c)
```

joue le même rôle que :

```
printf ("%c", c)
```

Son exécution est toutefois plus rapide, dans la mesure où elle ne fait pas appel au mécanisme d'analyse de format. Notez qu'en toute rigueur `putchar` n'est pas une vraie fonction mais une macro. Ses instructions (écrites en C) seront incorporées à votre programme par la directive :

```
#include <stdio.h>
```

Alors que cette directive était facultative pour `printf` (qui est une fonction), elle devient absolument nécessaire pour `putchar`. En son absence, l'éditeur de liens serait amené à rechercher une fonction `putchar` en bibliothèque et, ne la trouvant pas, il vous gratifierait d'un message d'erreur En toute rigueur, la fonction recherchée pourra porter un nom légèrement différent, par exemple `_putchar` ; c'est ce nom qui figurera dans le message d'erreur fourni par l'éditeur de liens.

## 2 Les possibilités de la fonction scanf

Nous avons déjà rencontré quelques exemples d'appels de `scanf`. Nous y avons notamment vu la nécessité de recourir à l'opérateur `&` pour désigner l'adresse de la variable (plus généralement de la *lvalue*) pour laquelle on souhaite lire une valeur. Vous avez pu remarquer que cette fonction possédait une certaine ressemblance avec `printf` et qu'en particulier elle faisait, elle aussi, appel à des « codes de format ».

Cependant, ces ressemblances masquent également des différences assez importantes au niveau :

- de la signification des codes de format. Certains codes correspondront à des types différents, suivant qu'ils sont employés avec `printf` ou avec `scanf` ;
- de l'interprétation des caractères du format qui ne font pas partie d'un code de format.

Ici, nous allons vous montrer le fonctionnement de `scanf`. Comme nous l'avons fait pour `printf`, nous vous présenterons d'abord les principaux codes de conversion. (Le paragraphe 1.2 de l'annexe vous en fournira un panorama complet). Là encore, nous avons également mentionné les codes de conversion relatifs aux chaînes et aux entiers non signés.

En revanche, compte tenu de la complexité de `scanf`, nous vous en exposerons les différentes possibilités de façon progressive, à l'aide d'exemples. Notamment, ce n'est qu'à la fin de ce chapitre que vous serez en mesure de connaître toutes les conséquences de données incorrectes.

## 2.1 Les principaux codes de conversion de `scanf`

Pour chaque code de conversion, nous précisons le type de la *lvalue* correspondante.

| | |
|---|---|
| **c** | `char` |
| **d** | `int` |
| **u** | `unsigned int` |
| **hd** | `short int` |
| **hu** | `unsigned short` |
| **ld** | `long int` |
| **lu** | `unsigned long` |
| **f** ou **e** | `loat` écrit indifféremment dans l'une des deux notations : décimale (éventuellement sans point, c'est-à-dire comme un entier) ou exponentielle (avec la lettre `e` ou `E`) |
| **lf** ou **le** | `double` avec la même présentation que ci-dessus |
| **s** | chaîne de caractères dont on fournit l'adresse (notion qui sera étudiée ultérieurement) |

**Remarque**

Contrairement à ce qui se passait pour `printf`, il ne peut plus y avoir ici de conversion automatique puisque l'argument transmis à `scanf` est l'adresse d'un emplacement mémoire. C'est ce qui justifie l'existence d'un code `hd` par exemple pour le type `short` ou encore celle des codes `lf` et *le* pour le type `double`.

## 2.2 Premières notions de tampon et de séparateurs

Lorsque `scanf` attend que vous lui fournissiez des données, l'information frappée au clavier est rangée temporairement dans l'emplacement mémoire nommé « tampon ». Ce dernier est exploré, caractère par caractère par `scanf`, au fur et à mesure des besoins. Il existe un pointeur qui précise quel est le prochain caractère à prendre en compte.

D'autre part, certains caractères dits « séparateurs » (ou « espaces blancs ») jouent un rôle particulier dans les données. Les deux principaux sont l'espace et la fin de ligne (`\n`). Il en existe trois autres d'un usage beaucoup moins fréquent : la tabulation horizontale (`\t`), la tabulation verticale (`\v`) et le changement de page (`\f`).

## 2.3 Les premières règles utilisées par `scanf`

Les codes de format correspondant à un nombre (c'est-à-dire tous ceux de la liste précédente, excepté `%c` et `%s`) entraînent d'abord l'avancement éventuel du pointeur jusqu'au premier caractère différent d'un séparateur. Puis `scanf` prend en compte tous les caractères suivants jusqu'à la rencontre d'un séparateur (en y plaçant le pointeur), du moins lorsque aucun gabarit

n'est précisé (comme nous apprendrons à le faire dans le paragraphe 2.4) et qu'aucun caractère invalide n'est présent dans la donnée (nous y reviendrons au paragraphe 2.6).

Quant au code de format `%c`, il entraîne la prise en compte du caractère désigné par le pointeur (même s'il s'agit d'un séparateur comme espace ou fin de ligne), et le pointeur est simplement avancé sur le caractère suivant du tampon.

Voici quelques exemples dans lesquels nous supposons que `n` et `p` sont de type `int`, tandis que `c` est de type `char`. Nous fournissons, pour chaque appel de `scanf`, des exemples de réponses possibles (`^` désigne un espace et `@` une fin de ligne) et, en regard, les valeurs effectivement lues.

```
scanf ("%d%d", &n, &p) ;
      12^25@              n = 12      p = 25
      ^12^^25^^@          n = 12      p = 25

      12@
      @
      ^25@                n = 12      p = 25
```

```
scanf ("%c%d", &c, &n) ;
      a25@                c = 'a'     n = 25
      a^^25@              c = 'a'     n = 25
```

```
scanf ("%d%c", &n, &c) ;
      12 a@               n = 12      c = ' '
```

Notez que, dans ce cas, on obtient bien le caractère « espace » dans `c`. Nous verrons dans le paragraphe 2.5 comment imposer à `scanf` de sauter quand même les espaces dans ce cas.

**Remarque**

Le code de format précise la nature du travail à effectuer pour transcoder une partie de l'information frappée au clavier, laquelle n'est en fait qu'une suite de caractères (codés chacun sur un octet) pour fabriquer la valeur (binaire) de la variable correspondante. Par exemple, `%d` entraîne en quelque sorte une double conversion : suite de caractères -> nombre écrit en décimal -> nombre codé en binaire ; la première conversion revient à faire correspondre un nombre entre 0 et 9 à un caractère représentant un chiffre. En revanche, le code `%c` demande simplement de ne rien faire puisqu'il suffit de recopier tel quel l'octet contenant le caractère concerné.

## 2.4 Imposition d'un gabarit maximal

Comme dans les codes de format de `printf`, on peut, dans un code de format de `scanf`, préciser un gabarit. Dans ce cas, le traitement d'un code de format s'interrompt soit à la rencontre d'un séparateur, soit lorsque le nombre de caractères indiqués a été atteint (attention, les séparateurs éventuellement sautés auparavant ne sont pas comptabilisés !).

Voici un exemple :

```
scanf ("%3d%3d", &n, &p)
      12^25@                    n = 12   p = 25

      ^^^^^12345@               n = 123  p = 45

      12@
      25@                       n = 12   p = 25
```

## 2.5 Rôle d'un espace dans le format

Un espace entre deux codes de format demande à `scanf` de faire avancer le pointeur au prochain caractère différent d'un séparateur. Notez que c'est déjà ce qui se passe lorsque l'on a affaire à un code de format correspondant à un type numérique. En revanche, cela n'était pas le cas pour les caractères, comme nous l'avons vu au paragraphe 2.3.

Voici un exemple :

```
scanf ("%d^%c", &n, &c) ;    /* ^ désigne un espace         */
                             /* %d^%c est différent de %d%c */
      12^a@              n = 12   c = 'a'
      12^^^a@            n = 12   c = 'a'
      12@a@              n = 12   c = 'a'
```

## 2.6 Cas où un caractère invalide apparaît dans une donnée

Voyez cet exemple, accompagné des valeurs obtenues dans les variables concernées :

```
scanf ("%d^%c", &n, &c) ;     /* ^ désigne un espace */
      12a@               n = 12   c = 'a'
```

Ce cas fait intervenir un mécanisme que nous n'avons pas encore rencontré. Il s'agit d'un troisième critère d'arrêt du traitement d'un code format (les deux premiers étaient : rencontre d'un séparateur ou gabarit atteint).

Ici, lors du traitement du code `%d`, `scanf` rencontre les caractères `1`, puis `2`, puis `a`. Ce caractère `a` ne convenant pas à la fabrication d'une valeur entière, `scanf` interrompt son exploration et fournit donc la valeur `12` pour `n`. L'espace qui suit `%d` dans le format n'a aucun effet puisque le caractère courant est le caractère `a` (différent d'un séparateur). Le traitement du code suivant, c'est-à-dire `%c`, amène `scanf` à prendre ce caractère courant (`a`) et à l'affecter à la variable `c`.

D'une manière générale, dans le traitement d'un code de format, `scanf` arrête son exploration du tampon dès que l'une des trois conditions est satisfaite :

- rencontre d'un caractère séparateur,

- gabarit maximal atteint (s'il y en a un de spécifié),
- rencontre d'un caractère invalide, par rapport à l'usage qu'on veut en faire (par exemple un point pour un entier, une lettre autre que `E` ou `e` pour un flottant,...). Notez bien l'aspect relatif de cette notion de caractère invalide.

## 2.7 Arrêt prématuré de scanf

Voyez cet exemple, dans lequel nous utilisons, pour la première fois, la valeur de retour de la fonction `scanf`.

```
compte = scanf ("%d^%d^%c", &n, &p, &c) ;     /* ^ désigne un espace */
 12^25^b@    n = 12        p = 25        c = 'b'        compte = 3
 12b@        n = 12        p inchangé    c inchangé     compte = 1
 b@          n indéfini    p inchangé    c inchangé     compte = 0
```

La valeur fournie par `scanf` n'est pas comparable à celle fournie par `printf` puisqu'il s'agit cette fois du **nombre de valeurs convenablement lues**. Ainsi, dans le premier cas, il n'est pas surprenant de constater que cette valeur est égale à `3`.

En revanche, dans le deuxième cas, le caractère `b` a interrompu le traitement du premier code `%d`. Dans le traitement du deuxième code (`%d`), `scanf` a rencontré d'emblée ce caractère `b`, toujours invalide pour une valeur numérique. Dans ces conditions, `scanf` se trouve dans l'incapacité d'attribuer une valeur à `p` (puisque ici, contrairement à ce qui s'est passé pour `n`, elle ne dispose d'aucun caractère correct). Dans un tel cas, `scanf` **s'interrompt sans chercher à lire d'autres valeurs** et fournit, en retour, le nombre de valeurs correctement lues jusqu'ici, c'est-à-dire `1`. Les valeurs de `p` et de `c` restent inchangées (éventuellement indéfinies).

Dans le troisième cas, le même mécanisme d'arrêt prématuré se produit dès le traitement du premier code de format, et le nombre de valeurs correctement lues est `0`.

**Ne confondez pas cet arrêt prématuré de *scanf* avec le troisième critère d'arrêt de traitement d'un code de format.** En effet, les deux situations possèdent bien la même cause (un caractère invalide par rapport à l'usage que l'on souhaite en faire), mais seul le cas où `scanf` n'est pas en mesure de fabriquer une valeur conduit à l'arrêt prématuré.

**Remarque**

Ici, nous avons vu la signification d'un espace introduit entre deux codes de format. En toute rigueur, vous pouvez introduire à un tel endroit n'importe quel caractère de votre choix. Dans ce cas, sachez que lorsque `scanf` rencontre un caractère (`x` par exemple) dans le format, il le compare avec le caractère courant (celui désigné par le pointeur) du tampon. S'ils sont égaux, il poursuit son travail (après avoir avancé le pointeur) mais, dans le cas contraire, il y a **arrêt prématuré**. Une telle possibilité ne doit toutefois être réservée qu'à des cas bien particuliers.

## 2.8 La syntaxe de scanf

D'une manière générale, l'appel de `scanf` se présente ainsi :

*La fonction* `scanf`

```
scanf (format, liste_d_adresses)
```

- `format` :
  - constante chaîne (entre " "),
  - pointeur sur une chaîne de caractères (cette notion sera étudiée ultérieurement).
- `liste_d_adresses` : liste de *lvalue*, séparées par des virgules, d'un type en accord avec le code de format correspondant.

## 2.9 Problèmes de synchronisation entre l'écran et le clavier

Voyez cet exemple de programme accompagné de son exécution alors que nous avons répondu :

```
12^25@
```

à la première question posée.

*L'écran et le clavier semblent mal synchronisés*

```
#include <stdio.h>
main()
{
    int n, p ;
    printf ("donnez une valeur pour n : ") ;
    scanf ("%d", &n) ;
    printf ("merci pour %d\n", n) ;
    printf ("donnez une valeur pour p : ") ;
    scanf ("%d", &p) ;
    printf ("merci pour %d", p) ;
}
```

```
donnez une valeur pour n : 12 25
merci pour 12
donnez une valeur pour p : merci pour 25
```

Vous constatez que la seconde question (donnez une valeur pour `p`) est apparue à l'écran, mais le programme n'a pas attendu que vous frappiez votre réponse pour vous afficher la suite. Vous notez alors qu'il a bien pris pour `p` la seconde valeur entrée au préalable, à savoir `25`.

En fait, comme nous l'avons vu, les informations frappées au clavier ne sont pas traitées instantanément par `scanf` mais mémorisées dans un tampon. Jusqu'ici, cependant, nous n'avions pas précisé quand `scanf` s'arrêtait de mémoriser pour commencer à traiter. Il le fait tout naturellement à la rencontre d'un caractère de fin de ligne généré par la frappe de la touche « return », dont le rôle est aussi classiquement celui d'une validation. Notez que, bien qu'il joue le rôle d'une validation, ce caractère de fin de ligne est quand même recopié dans le tampon ; il pourra donc éventuellement être lu en tant que tel.

L'élément nouveau réside donc dans le fait que `scanf` reçoit une information découpée en lignes (nous appelons ainsi une suite de caractères terminée par une fin de ligne). Tant que son traitement n'est pas terminé, elle attend une nouvelle ligne (c'est d'ailleurs ce qui se produisait dans notre premier exemple dans lequel nous commencions par frapper « return »).

Par contre, lorsque son traitement est terminé, s'il existe une partie de ligne non encore utilisée, celle-ci est conservée pour une prochaine lecture.

Autrement dit, le tampon n'est pas vidé à chaque nouvel appel de `scanf`. C'est ce qui explique le comportement du programme précédent.

## 2.10 En cas d'erreur

Dans le cas de `printf`, la source unique d'erreur résidait dans les fautes de programmation. Dans le cas de `scanf`, en revanche, il peut s'agir, non seulement d'une faute de programmation, mais également d'une mauvaise réponse de l'utilisateur.

### *2.10.1 Erreurs de programmation*

Comme dans le cas de `printf`, ces erreurs peuvent être de deux types :

**a) Code de format en désaccord avec le type de l'expression**

Si le code de format, bien qu'erroné, correspond à un type de longueur égale à celle de la *lvalue* mentionnée dans la liste, les conséquences se limitent, là encore, à l'introduction d'une mauvaise valeur. Si, en revanche, la *lvalue* a une taille inférieure à celle correspondant au type mentionné dans le code format, il y aura écrasement d'un emplacement mémoire consécutif à cette *lvalue*. Les conséquences en sont difficilement prévisibles.

**b) Nombre de codes de format différent du nombre d'éléments de la liste**

Comme dans le cas de `printf`, il faut savoir que `scanf` **cherche toujours à satisfaire le contenu du format**. Les conséquences sont limitées dans le cas où le format comporte moins de codes que la liste ; ainsi, dans cette instruction, on ne cherchera à lire que la valeur de `n` :

```
scanf ("%d", &n, &p) ;
```

En revanche, dans le cas où le format comporte plus de codes que la liste, on cherchera à affecter des valeurs à des emplacements (presque) aléatoires de la mémoire. Là encore, les conséquences en seront pratiquement imprévisibles.

### *2.10.2 Mauvaise réponse de l'utilisateur*

Nous avons déjà vu ce qui se passait lorsque l'utilisateur fournissait trop ou trop peu d'information par rapport à ce qu'attendait `scanf`.

De même, nous avons vu comment, en cas de rencontre d'un caractère invalide, il y avait arrêt prématuré. Dans ce cas, il faut bien voir que ce caractère non exploité reste dans le tampon pour une prochaine fois. Cela peut conduire à des situations assez cocasses telles que celle qui est présentée dans cet exemple (l'impression de `^C` représente, dans l'environnement utilisé, une interruption du programme par l'utilisateur) :

*Boucle infinie sur un caractère invalide*

```
main()
{
   int n ;
   do
      { printf ("donnez un nombre : ") ;
        scanf ("%d", &n) ;
        printf ("voici son carré : %d\n", n*n) ;
      }
   while (n) ;
}
```

```
donnez un nombre : 12
voici son carré : 144
donnez un nombre : &
voici son carré : 144
donnez un nombre : voici son carré : 144
donnez un nombre : voici son carré : 144
donnez un nombre : voici son carré : 144
donnez un nombre : voici son carré : 144
 ^C
```

Fort heureusement, il existe un remède à cette situation. Nous ne pourrons vous l'exposer complètement que lorsque nous aurons étudié les chaînes de caractères.

## 2.11 La macro getchar

L'expression :

```
c = getchar()
```

joue le même rôle que :

```
scanf ("%c", &c)
```

tout en étant plus rapide puisque ne faisant pas appel au mécanisme d'analyse d'un format.

Notez bien que `getchar` **utilise le même tampon (image d'une ligne) que** `scanf`**.**

En toute rigueur, `getchar` est une macro (comme `putchar`) dont les instructions figurent dans `stdio.h`. Là encore, l'omission d'une instruction `#include` appropriée conduit à une erreur à l'édition de liens.

# Exercices

Tous ces exercices sont corrigés en fin de volume.

**1)** Quels seront les résultats fournis par ce programme ?

```
#include <stdio.h>
main ()
{   int n = 543 ;
    int p = 5 ;
    float x = 34.5678;
    printf ("A : %d %f\n", n, x) ;
    printf ("B : %4d %10f\n", n, x) ;
    printf ("C : %2d %3f\n", n, x) ;
    printf ("D : %10.3f %10.3e\n", x, x) ;
    printf ("E : %*d\n", p, n) ;
    printf ("F : %*.*f\n", 12, 5, x) ;
}
```

**2)** Quelles seront les valeurs lues dans les variables *n* et *p* (de type *int*), par l'instruction suivante ?

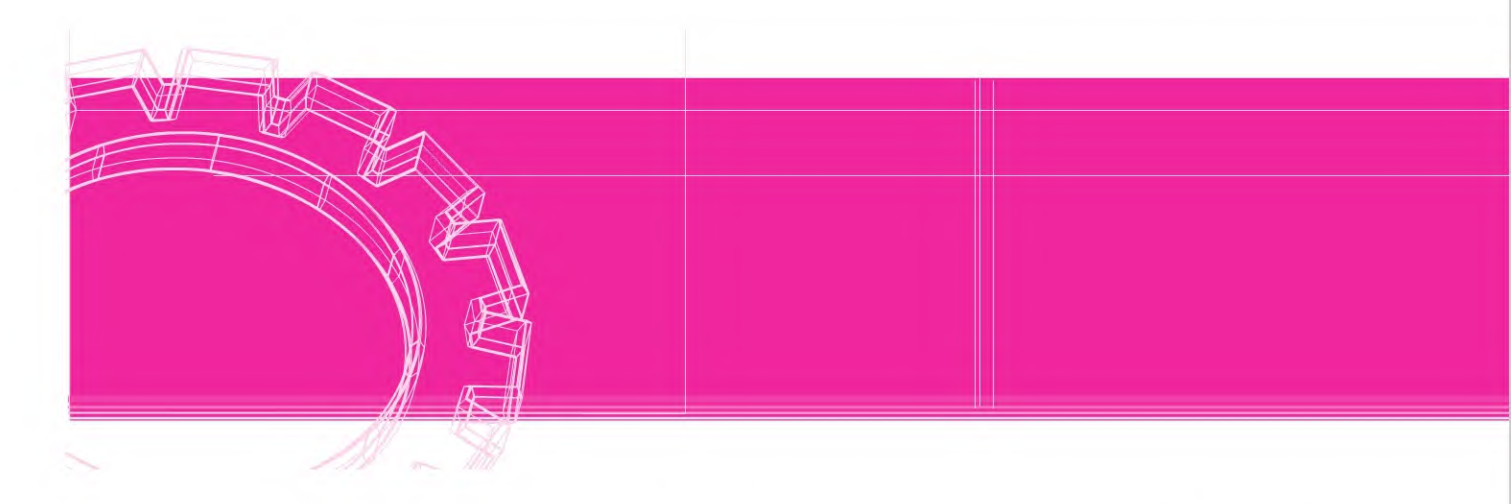
```
scanf ("%4d %2d", &n, &p) ;
```

lorsqu'on lui fournit les données suivantes (le symbole ^ représente un espace et le symbole @ représente une fin de ligne, c'est-à-dire une validation) ?

```
a) 12^45@
b) 123456@
c) 123456^7@
d) 1^458@
e) ^^^4567^^8912@
```

# Chapitre 5

# Les instructions de contrôle



A priori, dans un programme, les instructions sont exécutées séquentiellement, c'est-à-dire dans l'ordre où elles apparaissent. Or la puissance et le « comportement intelligent » d'un programme proviennent essentiellement :

- de la possibilité d'effectuer des **choix**, de se comporter différemment suivant les circonstances (celles-ci pouvant être, par exemple, une réponse de l'utilisateur, un résultat de calcul...),
- de la possibilité d'effectuer des **boucles**, autrement dit de répéter plusieurs fois un ensemble donné d'instructions.

Tous les langages disposent d'instructions, nommées *instructions de contrôle*, permettant de réaliser ces choix ou ces boucles. Suivant le cas, celles-ci peuvent être :

- basées essentiellement sur la notion de branchement (conditionnel ou inconditionnel) ; c'était le cas, par exemple, des premiers Basic,
- ou, au contraire, traduire fidèlement les structures fondamentales de la programmation structurée ; cela était le cas, par exemple, du langage Pascal bien que, en toute rigueur, ce dernier dispose d'une instruction de branchement inconditionnel GOTO.

Sur ce point, le langage C est quelque peu hybride. En effet d'une part, il dispose d'instructions structurées permettant de réaliser :

- des choix : instructions **`if...else`** et **`switch`**,
- des boucles : instructions **`do...while`**, **`while`** et **`for`**.

Mais, d'autre part, la notion de branchement n'en est pas totalement absente puisque, comme nous le verrons :

- il dispose d'instructions de branchement inconditionnel : **`goto`**, **`break`** et **`continue`**,
- l'instruction `switch` est en fait intermédiaire entre un choix multiple parfaitement structuré (comme dans Pascal) et un aiguillage multiple (comme dans Fortran).

Ce sont ces différentes instructions de contrôle du langage C que nous nous proposons d'étudier dans ce chapitre.

# 1 L'instruction `if`

Nous avons déjà rencontré des exemples d'instruction `if` et nous avons vu que cette dernière pouvait éventuellement faire intervenir un bloc. Précisons donc tout d'abord ce qu'est un bloc d'une manière générale.

## 1.1 Blocs d'instructions

Un bloc est une suite d'instructions placées entre `{` et `}`. Les instructions figurant dans un bloc sont absolument quelconques. Il peut s'agir aussi bien d'instructions simples (terminées par un point-virgule) que d'instructions structurées (choix, boucles) lesquelles peuvent alors à leur tour renfermer d'autres blocs.

Rappelons qu'en C, la notion d'instruction est en quelque sorte récursive. Dans la description de la syntaxe des différentes instructions, nous serons souvent amené à mentionner ce terme d'**instruction**. Comme nous l'avons déjà noté, celui-ci désignera toujours n'importe quelle instruction C : **simple**, **structurée** ou un **bloc**.

Un bloc peut se réduire à une seule instruction, voire être vide. Voici deux exemples de blocs corrects :

```
{ }
{ i = 1 ; }
```

Le second bloc ne présente aucun intérêt en pratique puisqu'il pourra toujours être remplacé par l'instruction simple qu'il contient.

En revanche, nous verrons que le premier bloc (lequel pourrait a priori être remplacé par... rien) apportera une meilleure lisibilité dans le cas de boucles ayant un corps vide.

Notez encore que `{ ; }` est un bloc constitué d'une seule instruction vide, ce qui est syntaxiquement correct.

**Remarque**

**Important.** N'oubliez pas que toute instruction simple est toujours terminée par un point-virgule. Ainsi, ce bloc :

```
{ i = 5 ; k = 3 }
```

est incorrect car il manque un point-virgule à la fin de la seconde instruction.

D'autre part, un bloc joue le même rôle syntaxique qu'une instruction simple (point-virgule compris). Évitez donc d'ajouter des points-virgules intempestifs à la suite d'un bloc.

## 1.2 Syntaxe de l'instruction `if`

Le mot `else` et l'instruction qu'il introduit sont facultatifs, de sorte que cette instruction `if` présente deux formes.

*L'instruction* if

```
if (expression)                          if (expression)
        instruction_1                            instruction_1
    else
        instruction_2
```

- `expression` *:* expression quelconque
- `instruction_1` et `instruction_2` : instructions quelconques, c'est-à-dire :
  - simple (terminée par un point-virgule),
  - bloc,
  - instruction structurée.

**Remarque**

La syntaxe de cette instruction n'impose en soi aucun point-virgule, si ce n'est ceux qui terminent naturellement les instructions simples qui y figurent.

## 1.3 Exemples

L'expression conditionnant le choix est quelconque. La richesse de la notion d'expression en C fait que celle-ci peut elle-même réaliser certaines actions. Ainsi :

```
if ( ++i < limite)  printf ("OK") ;
```

est équivalent à :

```
i = i + 1 ;
if ( i < limite )  printf ("OK") ;
```

Par ailleurs :

```
if ( i++ < limite ) ......
```

est équivalent à :

```
i = i + 1 ;
if ( i-1 < limite ) ......
```

De même :

```
if ( ( c=getchar() ) != '\n' ) ......
```

peut remplacer :

```
c = getchar() ;
if ( c != '\n' ) ......
```

En revanche :

```
if ( ++i<max && ( (c=getchar()) != '\n') ) ......
```

n'est **pas équivalent à :**

```
++i ;
c = getchar() ;
if ( i<max && ( c!= '\n' ) ) ......
```

car, comme nous l'avons déjà dit, l'opérateur `&&` n'évalue son second opérande que lorsque cela est nécessaire. Autrement dit, dans la première formulation, l'expression :

```
c = getchar()
```

n'est pas évaluée lorsque la condition `++i<max` est fausse ; elle l'est, en revanche, dans la deuxième formulation.

## 1.4 Imbrication des instructions `if`

Nous avons déjà mentionné que les instructions figurant dans chaque partie du choix d'une instruction pouvaient être absolument quelconques. En particulier, elles peuvent, à leur tour, renfermer d'autres instructions `if`. Or, compte tenu de ce que cette instruction peut comporter ou ne pas comporter de `else`, il existe certaines situations où une ambiguïté apparaît. C'est le cas dans cet exemple :

```
if (a<=b)  if (b<=c)  printf ("ordonné") ;
                else printf ("non ordonné") ;
```

Est-il interprété comme le suggère cette présentation ?

```
if (a<=b)  if (b<=c) printf ("ordonné") ;
     else  printf ("non ordonné") ;
```

ou bien comme le suggère celle-ci ?

```
if (a<=b) if (b<=c) printf ("ordonné") ;
               else printf ("non ordonné") ;
```

La première interprétation conduirait à afficher `"non ordonné"` lorsque la condition `a<=b` est fausse, tandis que la seconde n'afficherait rien dans ce cas. La règle adoptée par le langage C pour lever une telle ambiguïté est la suivante :

**Un `else` se rapporte toujours au dernier `if` rencontré auquel un `else` n'a pas encore été attribué.**

Dans notre exemple, c'est la seconde présentation qui suggère le mieux ce qui se passe.

Voici un exemple d'utilisation de `if` imbriqués. Il s'agit d'un programme de facturation avec remise. Il lit en donnée un simple prix hors taxes et calcule le prix TTC correspondant (avec un taux de TVA constant de 18,6 %). Il établit ensuite une remise dont le taux dépend de la valeur ainsi obtenue, à savoir :

- 0 % pour un montant inférieur à 1 000 F
- 1 % pour un montant supérieur ou égal à 1 000 F et inférieur à 2 000 F
- 3 % pour un montant supérieur ou égal à 2 000 F et inférieur à 5 000 F
- 5 % pour un montant supérieur ou égal à 5 000 F

Ce programme est accompagné de deux exemples d'exécution.

*Exemple de* if *imbriqués : facturation avec remise*

```
#define TAUX_TVA  18.6
main()
{
   double ht, ttc, net, tauxr, remise ;
   printf("donnez le prix hors taxes : ") ;
   scanf ("%lf", &ht) ;

   ttc = ht * ( 1. + TAUX_TVA/100.) ;
   if ( ttc < 1000.)            tauxr = 0 ;
      else if ( ttc < 2000 )    tauxr = 1. ;
         else if ( ttc < 5000 ) tauxr = 3. ;
            else                tauxr = 5. ;

   remise = ttc * tauxr / 100. ;
   net = ttc - remise ;
   printf ("prix ttc     %10.2lf\n", ttc) ;
   printf ("remise       %10.2lf\n", remise) ;
   printf ("net à payer  %10.2lf\n", net) ;
}
```

*Exemple de* if *imbriqués : facturation avec remise (suite)*

```
donnez le prix hors taxes : 500
prix ttc         593.00
remise             0.00
net à payer      593.00
```

---

```
donnez le prix hors taxes : 4000
prix ttc        4744.00
remise           142.32
net à payer     4601.68
```

# 2 Instruction switch

## 2.1 Exemples d'introduction de l'instruction switch

### a) Premier exemple

Voyez ce premier exemple de programme accompagné de trois exemples d'exécution.

*Premier exemple d'instruction* switch

```
main()
{
   int n ;
   printf ("donnez un entier : ") ;
   scanf ("%d", &n) ;

   switch (n)
   {  case 0 : printf ("nul\n") ;
               break ;
      case 1 : printf ("un\n") ;
               break ;
      case 2 : printf ("deux\n") ;
               break ;
   }

   printf ("au revoir\n") ;
}
```

*Premier exemple d'instruction* switch *(suite)*

```
donnez un entier : 0
nul
au revoir
______
donnez un entier : 2
deux
au revoir
______
donnez un entier : 5
au revoir
```

L'instruction `switch` s'étend ici sur huit lignes (elle commence au mot `switch`). Son exécution se déroule comme suit. On commence tout d'abord par évaluer l'expression figurant après le mot `switch` (ici `n`). Puis, on recherche dans le *bloc* qui suit s'il existe une « *étiquette* » de la forme « *case x* » correspondant à la valeur ainsi obtenue. Si c'est le cas, on se branche à l'instruction figurant après cette étiquette. Dans le cas contraire, on passe à l'instruction qui suit le bloc.

Par exemple, quand `n` vaut `0`, on trouve effectivement une étiquette `case 0` et l'on exécute l'instruction correspondante, c'est-à-dire :

```
printf ("nul") ;
```

On passe ensuite, naturellement, à l'instruction suivante, à savoir, ici :

```
break ;
```

Celle-ci demande en fait de sortir du bloc. Notez bien que le rôle de cette instruction est fondamental. Voyez, à titre d'exemple, ce que produirait ce même programme en l'absence d'instructions `break` :

*Absence d'instructions* `break`

```
main()
{
   int n ;
   printf ("donnez un entier : ") ;
   scanf ("%d", &n) ;

   switch (n)
   {  case 0 : printf ("nul\n") ;
      case 1 : printf ("un\n") ;
      case 2 : printf ("deux\n") ;
   }
   printf ("au revoir\n") ;
}
```

*Absence d'instructions* `break` *(suite)*

```
donnez un entier : 0
nul
un
deux
au revoir
______
donnez un entier : 2
deux
au revoir
```

### b) Étiquette default

Il est possible d'utiliser le mot-clé **`default`** comme étiquette à laquelle le programme se branchera dans le cas où aucune valeur satisfaisante n'aura été rencontrée auparavant.

En voici un exemple :

*Étiquette* default

```
main()
{
   int n ;
   printf ("donnez un entier : ") ;
   scanf ("%d", &n) ;
   switch (n)
   {  case 0 :  printf ("nul\n") ;
                break ;
      case 1 :  printf ("un\n") ;
                break ;
      case 2 :  printf ("deux\n") ;
                break ;
      default : printf ("grand\n") ;
   }
   printf ("au revoir\n") ;
}
```

```
donnez un entier : 2
deux
au revoir
_______
donnez un entier : 25
grand
au revoir
```

### c) Exemple plus général

D'une manière générale, on peut trouver :

- plusieurs instructions à la suite d'une étiquette,
- des étiquettes sans instructions, c'est-à-dire, en définitive, plusieurs étiquettes successives (accompagnées de leurs deux-points).

Voyez cet exemple, dans lequel nous avons volontairement omis certains `break`.

*Exemple général d'instruction* switch

```
main()
{
   int n ;
   printf ("donnez un entier : ") ;
   scanf ("%d", &n) ;

   switch (n)
   {  case 0   : printf ("nul\n") ;
                 break ;
      case 1   :
      case 2   : printf ("petit\n") ;
      case 3   :
      case 4   :
      case 5   : printf ("moyen\n") ;
                 break ;
      default : printf ("grand\n") ;
   }
}
```

```
donnez un entier : 1
petit
moyen
______
donnez un entier : 4
moyen
______
donnez un entier : 25
grand
```

## 2.2 Syntaxe de l'instruction switch

*L'instruction* switch

```
switch (expression)
   { case constante_1 : [ suite_d'instructions_1 ]
     case constante_2 : [ suite_d'instructions_2 ]
                ..............
     case constante_n : [ suite_d'instructions_n ]
     [ default        :   suite_d'instructions   ]
   }
```

- `expression` : expression entière quelconque,
- `constante` : expression constante d'un type entier quelconque (`char` est accepté car il sera converti en `int`),
- `suite_d'instructions` : séquence d'instructions quelconques.

**Remarque**

Les crochets ( [ et ] ) signifient que ce qu'ils renferment est facultatif.

### *Commentaires :*

**1)** Il paraît normal que cette instruction limite les valeurs des étiquettes à des valeurs entières ; en effet, il ne faut pas oublier que la comparaison d'égalité de la valeur d'une expression flottante à celle d'une constante flottante est relativement aléatoire, compte tenu de la précision limitée des calculs. En revanche, il est possible d'employer des constantes de type caractère, étant donné qu'il y aura systématiquement conversion en `int`. Cela autorise des constructions du type :

```
switch(c)
  {  case 'a' : ......
     case 132 : .....
        ......
  }
```

où `c` est de type `char`, ou encore :

```
switch (n)
  {  case 'A' : .....
     case 559 : .....
        .......
  }
```

où `n` est du type `int`.

**2)** La syntaxe autorise des expressions constantes et non seulement des constantes. On nomme ainsi des expressions qui peuvent être évaluées lors de la compilation. Cela peut être, bien sûr, des expressions telles que :

```
5 + 2            3  * 8 - 2
```

mais l'intérêt en reste limité puisqu'il est alors toujours possible de faire le calcul soi-même.

Mais cela peut également faire appel à des symboles définis par la directive `#define`, comme dans cet exemple :

```
#define  LIMITE  20
    .....
switch (n)
   {  .....
      case LIMITE-1 : ......
      case LIMITE   : ......
      case LIMITE+1 : ......
   }
```

À la compilation, les expressions `LIMITE-1`, `LIMITE` et `LIMITE+1` seront effectivement remplacées par les valeurs `19`, `20` et `21`.

Cette façon de procéder permet un certain paramétrage des programmes. Ainsi, dans cet exemple, une modification de la valeur de `LIMITE` se résume à une seule intervention au niveau de la directive `#define`. Notez bien qu'une variable initialisée à `20` au sein du programme ne pourrait pas être utilisée puisque les étiquettes de l'instruction `switch` ne seraient plus des expressions constantes.

**3)** Les connaisseurs du Pascal trouveront que cette sélection réalisée par l'instruction `switch` est moins riche que celle offerte par l'instruction `CASE` dans la mesure où elle impose d'énumérer les différentes valeurs concernées. En aucun cas, on ne peut fournir un intervalle autrement qu'en citant chacune de ses valeurs.

# 3 L'instruction do... while

Abordons maintenant la première façon de réaliser une boucle en C, à savoir l'instruction `do... while`.

## 3.1 Exemple d'introduction de l'instruction do... while

*Exemple d'instruction* do... while

```
main()
{
   int n ;
   do
      { printf ("donnez un nb >0 : ") ;
        scanf ("%d", &n) ;
        printf ("vous avez fourni %d\n", n) ;
      }
   while (n<=0) ;
   printf ("réponse correcte") ;
}
```

```
donnez un nb >0 : -3
vous avez fourni -3
donnez un nb >0 : -9
vous avez fourni -9
donnez un nb >0 : 12
vous avez fourni 12
réponse correcte
```

L'instruction :

```
do  {  .......  }  while (n<=0) ;
```

répète l'instruction qu'elle contient (ici un bloc) tant que la condition mentionnée (`n<=0`) est vraie (c'est-à-dire, en C, non nulle). Autrement dit, ici, elle demande un nombre à l'utilisateur (en affichant la valeur lue) tant qu'il ne fournit pas une valeur positive.

On ne sait pas a priori combien de fois une telle boucle sera répétée. Toutefois, de par sa nature même, elle est toujours parcourue au moins une fois. En effet, la condition qui régit cette boucle n'est examinée qu'à la fin de chaque répétition (comme le suggère d'ailleurs le fait que la « partie `while` » figure en fin).

Notez bien que la sortie de boucle ne se fait qu'après un parcours complet de ses instructions et non dès que la condition mentionnée devient fausse. Ainsi, ici, même après que l'utilisateur a fourni une réponse convenable, il y a exécution de l'instruction d'affichage :

```
printf ("vous avez fourni %d", n) ;
```

## 3.2 Syntaxe de l'instruction `do... while`

*L'instruction* do... while

```
do     instruction

       while  (expression) ;
```

### *Commentaires*

**1)** Notez bien, d'une part la présence de **parenthèses** autour de l'expression qui régit la poursuite de la boucle, d'autre part la présence d'un **point-virgule** à la fin de cette instruction.

**2)** Lorsque l'instruction à répéter se limite à une seule instruction simple, n'omettez pas le point-virgule qui la termine. Ainsi :

```
 do c = getchar() while ( c != 'x') ;
```

est incorrecte. Il faut absolument écrire :

```
do c = getchar() ; while ( c != 'x') ;
```

**3)** N'oubliez pas que, là encore, l'expression suivant le mot `while` peut être aussi élaborée que vous le souhaitez et qu'elle permet ainsi de réaliser certaines actions. Nous en verrons quelques exemples dans le paragraphe suivant.

**4)** L'instruction à répéter peut être vide (mais quand même terminée par un point-virgule). Ces constructions sont correctes :

```
do ; while ( ... ) ;

do { } while ( ... ) ;
```

**5)** La construction :

```
do { } while (1) ;
```

représente une boucle infinie ; elle est syntaxiquement correcte, bien qu'elle ne présente en pratique aucun intérêt. En revanche :

```
do  instruction  while (1) ;
```

pourra présenter un intérêt dans la mesure où, comme nous le verrons, il sera possible d'en sortir éventuellement par une instruction `break`.

**6)** Si vous connaissez Pascal, vous remarquerez que cette instruction `do... while` correspond au `repeat... until` avec, cependant, une condition exprimée sous forme contraire.

## 3.3 Exemples

**a)** L'exemple proposé au paragraphe 3.1 peut également s'écrire :

```
do  { printf ("donnez un nb > 0 : ") ;
      scanf ("%d", &n) ;
    }
while ( printf("vous avez fourni %d", n), n<= 0 )
```

ou encore :

```
do  printf ("donnez un nb >0 : ") ;
while (scanf("%d", &n), printf ("vous avez fourni %d", n), n <= 0 ) ;
```

ou même :

```
do { }
while ( printf ("donnez un nb > 0 :"),  scanf ("%d", &n),
        printf ("vous avez fourni %d", n), n <= 0 ) ;
```

Notez bien que la condition de poursuite doit être la dernière expression évaluée, compte tenu du fonctionnement de l'opérateur séquentiel.

**b)** L'instruction :

```
do { } while ( (c=getchar()) != 'x' ) ;
```

lit des caractères au clavier jusqu'à ce qu'elle ait obtenu le caractère `x`. Elle est équivalente à :

```
do  c = getchar() ;
while ( c != 'x' ) ;
```

# 4 L'instruction `while`

Voyons maintenant la deuxième façon de réaliser une boucle conditionnelle, à savoir l'instruction `while`.

## 4.1 Exemple d'introduction de l'instruction while

*Exemple d'instruction* while

```
main()
{
   int n, som ;
   som = 0 ;
   while (som<100
      { printf ("donnez un nombre : ") ;
        scanf ("%d", &n) ;
        som += n ;
      }
   printf ("somme obtenue : %d", som) ;
}
```

```
donnez un nombre : 15
donnez un nombre : 25
donnez un nombre : 12
donnez un nombre : 60
somme obtenue : 112
```

La construction :

```
while (som<100)
```

répète l'instruction qui suit (ici un bloc) tant que la condition mentionnée est vraie (différente de zéro), comme le ferait `do... while`. En revanche, cette fois, la condition de poursuite est examinée **avant** chaque parcours de la boucle et non après. Ainsi, contrairement à ce qui se passait avec `do... while`, une telle boucle peut très bien n'être parcourue aucune fois si la condition est fausse dès qu'on l'aborde (ce qui n'est pas le cas ici).

## 4.2 Syntaxe de l'instruction while

*L'instruction* while

```
while  (expression)
     instruction
```

### *Commentaires*

**1)** Là encore, notez bien la présence de parenthèses pour délimiter la condition de poursuite. Remarquez que, par contre, la syntaxe n'impose aucun point-virgule de fin (il s'en trouvera naturellement un à la fin de l'instruction qui suit si celle-ci est simple).

**2)** L'expression utilisée comme condition de poursuite est évaluée avant le premier tour de boucle. Il est donc nécessaire que sa valeur soit définie à ce moment.

**3)** Lorsque la condition de poursuite est une expression qui fait appel à l'opérateur séquentiel, n'oubliez pas qu'alors toutes les expressions qui la constituent seront évaluées avant le test de poursuite de la boucle. Ainsi, cette construction :

```
while ( printf ("donnez un nombre : ") , scanf ("%d", &n), som<=100)
            som += n ;
```

n'est pas équivalente à celle de l'exemple d'introduction.

**4)** La construction :

```
    while ( expression1, expression2 ) ;
```

est équivalente à :

```
    do expression1
        while ( expression2 ) ;
```

Par exemple, ces deux instructions sont équivalentes :

```
while ( (c=getchar()) != 'x' ) { }
do { } while ( (c=getchar()) != 'x' ) ;
```

# 5 L'instruction for

Étudions maintenant la dernière instruction permettant de réaliser des boucles, à savoir l'instruction for.

## 5.1 Exemple d'introduction de l'instruction for

Considérez ce programme :

*Exemple d'instruction* for

```
main()
{
   int i ;
   for ( i=1 ; i<=5 ; i++ )
      {  printf ("bonjour ") ;
         printf ("%d fois\n", i) ;
      }
}
```

*Exemple d'instruction* for *(suite)*

```
bonjour 1 fois
bonjour 2 fois
bonjour 3 fois
bonjour 4 fois
bonjour 5 fois
```

La ligne :

```
for ( i=1 ; i<=5 ; i++ )
```

comporte en fait trois expressions. La première est évaluée (une seule fois) avant d'entrer dans la boucle. La deuxième conditionne la poursuite de la boucle. Elle est évaluée **avant** chaque parcours. La troisième, enfin, est évaluée à la fin de chaque parcours.

Le programme précédent est équivalent au suivant :

*Remplacement d' une boucle* for *par une boucle* while

```
main()
{
   int i ;
   i = 1 ;
   while (i<=5)
      {  printf ("bonjour ") ;
         printf ("%d fois\n", i) ;
         i++ ;
      }
}
```

```
bonjour 1 fois
bonjour 2 fois
bonjour 3 fois
bonjour 4 fois
bonjour 5 fois
```

Là encore, la généralité de la notion d'expression en C fait que ce qui était expression dans la première formulation (`for`) devient instruction dans la seconde (`while`).

## 5.2 Syntaxe de l'instruction for

*L'instruction* for

```
for  ( [ expression_1 ] ; [ expression_2 ] ; [ expression_3 ] )
              instruction
```

Les crochets [ et ] signifient que leur contenu est facultatif.

### *Commentaires*

**1)** D'une manière générale, nous pouvons dire que :

```
for ( expression_1 ; expression_2 ; expression_3) instruction
```

est équivalent à :

```
expression_1 ;
while (expression_2)
   {  instruction
      expression_3 ;
   }
```

**2)** Chacune des trois expressions est facultative. Ainsi, ces constructions sont équivalentes à l'instruction `for` de notre premier exemple de programme :

```
i = 1 ;
for ( ; i<=5 ; i++ )  { printf ("bonjour ") ;
                        printf ("%d fois\n", i) ;
                      }

i = 1 ;
for ( ; i<=5 ; )    { printf ("bonjour ") ;
                      printf ("%d fois\n", i) ;
                      i++ ;
                    }
```

**3)** Lorsque `l'expression_2` est absente, elle est considérée comme vraie.

**4)** Là encore, la richesse de la notion d'expression en C permet de grouper plusieurs actions dans une expression. Ainsi :

```
for ( i=0, j=1, k=5 ; ... ; ... )
```

est équivalent à :

```
j=1 ; k=5 ;
for ( i=0 ; ... ; ... )
```

ou encore à :

```
i=0 ; j=1 ; k=5 ;
for ( ; ... ; ...)
```

De même :

```
for ( i=1 ; i <= 5 ; printf("fin de tour"), i++ )  { instructions }
```

est équivalent à :

```
for ( i=1 ; i<=5 ; i++ )
      {  instructions
         printf ("fin de tour") ;
      }
```

En revanche :

```
for ( i=1, printf("on commence") ; printf("début de tour"), i<=5 ; i++)
         { instructions }
```

n'est pas équivalent à :

```
printf ("on commence") ;
for ( i=1 ; i<=5 ; i++ )
      { printf ("début de tour") ;
        instructions
      }
```

car, dans la première construction, le message `début de tour` est affiché après le dernier tour tandis qu'il ne l'est pas dans la seconde construction.

**5)** Les deux constructions :

```
for ( ; ; ) ;
for ( ; ; ) { }
```

sont syntaxiquement correctes. Elles représentent des boucles infinies de corps vide (n'oubliez pas que, lorsque la seconde expression est absente, elle est considérée comme vraie). En pratique, elles ne présentent aucun intérêt.

En revanche, cette construction

```
for ( ; ; )  instruction
```

est une boucle a priori infinie dont on pourra éventuellement sortir par une instruction `break` (comme nous le verrons dans le paragraphe suivant).

**Remarque**

Contrairement à ce qui se passe dans beaucoup de langages, les trois instructions de boucle du langage C sont des boucles conditionnelles. En effet, l'instruction `for`, basée sur une condition, n'est pas l'équivalent strict de la « répétition avec compteur » (ce qui est le cas du `for` du Pascal et du Basic ou du `do` du Fortran), même si c'est généralement celle que l'on utilise en C pour jouer un tel rôle.

# 6 Les instructions de branchement inconditionnel : `break`, `continue` et `goto`

Ces trois instructions fournissent des possibilités diverses de branchement inconditionnel. Les deux premières s'emploient principalement au sein de boucles tandis que la dernière est d'un usage libre mais peu répandu, à partir du moment où l'on cherche à structurer quelque peu ses programmes.

## 6.1 L'instruction `break`

Nous avons déjà vu le rôle de `break` au sein du bloc régi par une instruction `switch`.

Le langage C autorise également l'emploi de cette instruction dans une boucle. Dans ce cas, elle sert à interrompre le déroulement de la boucle, en passant à l'instruction qui suit cette boucle. Bien entendu, cette instruction n'a d'intérêt que si son exécution est conditionnée par un choix ; dans le cas contraire, en effet, elle serait exécutée dès le premier tour de boucle, ce qui rendrait la boucle inutile.

Voici un exemple montrant le fonctionnement de `break` :

*Exemple d'instruction* break

```
main()
{
   int i ;
   for ( i=1 ; i<=10 ; i++ )
      { printf ("début tour %d\n", i) ;
        printf ("bonjour\n")
        if ( i==3 ) break ;
        printf ("fin tour %d\n", i) ;
      }
   printf ("après la boucle") ;
}
```

*Exemple d'instruction* break *(suite)*

```
début tour 1
bonjour
fin tour 1
début tour 2
bonjour
fin tour 2
début tour 3
bonjour
après la boucle
```

**Remarque** En cas de boucles imbriquées, `break` fait sortir de la boucle la plus interne. De même si `break` apparaît dans un `switch` imbriqué dans une boucle, elle ne fait sortir que du `switch`.

## 6.2 L'instruction `continue`

L'instruction `continue`, quant à elle, permet de passer prématurément au tour de boucle suivant. En voici un premier exemple avec `for` :

*Exemple d'instruction* continue *dans une boucle* for

```
main()
{  int i ;
   for ( i=1 ; i<=5 ; i++ )
      { printf ("début tour %d\n", i) ;
        if (i<4) continue ;
        printf ("bonjour\n") ;
      }
}
```

```
début tour 1
début tour 2
début tour 3
début tour 4
bonjour
début tour 5
bonjour
```

Et voici un second exemple avec `do... while` :

*Exemple d'instruction* continue *dans une boucle* do... while

```
main()
{  int n ;
   do
      { printf ("donnez un nb>0 : ") ;
        scanf ("%d", &n) ;
        if (n<0) { printf ("svp >0\n") ;
                   continue ;
                 }
        printf ("son carré est : %d\n", n*n) ;
      }
   while(n) ;
}
```

```
donnez un nb>0 : 4
son carré est : 16
donnez un nb>0 : -5
svp >0
donnez un nb>0 : 2
son carré est : 4
donnez un nb>0 : 0
son carré est : 0
```

**Remarques**

Lorsqu'elle est utilisée dans une boucle `for`, cette instruction `continue` effectue bien un branchement sur l'évaluation de l'expression de fin de parcours de boucle (nommée `expression_2` dans la présentation de sa syntaxe), et non après.

En cas de boucles imbriquées, l'instruction *continue* ne concerne que la boucle la plus interne.

## 6.3 L'instruction goto

Elle permet classiquement le branchement en un emplacement quelconque du programme. Voyez cet exemple qui simule, dans une boucle `for`, l'instruction `break` à l'aide de l'instruction `goto` (ce programme fournit les mêmes résultats que celui présenté comme exemple de l'instruction `break`).

*Exemple d'instruction* goto

```
main()
{
   int i ;
   for ( i=1 ; i<=10 ; i++ )
      { printf ("début tour %d\n", i) ;
        printf ("bonjour\n") ;
        if ( i==3 ) goto sortie ;
        printf ("fin tour %d\n", i) ;
      }
   sortie : printf ("après la boucle") ;
}
```

```
début tour 1
bonjour
fin tour 1
début tour 2
bonjour
fin tour 2
début tour 3
bonjour
après la boucle
```

# Exercices

Tous ces exercices sont corrigés en fin de volume.

**1)** Soit le petit programme suivant :

```
#include <stdio.h>
main()
{
   int i, n, som ;
   som = 0 ;
   for (i=0 ; i<4 ; i++)
      { printf ("donnez un entier ") ;
        scanf ("%d", &n) ;
        som += n ;
      }
   printf ("Somme : %d\n", som) ;
}
```

Écrire un programme réalisant exactement la même chose, en employant, à la place de l'instruction `for` :

- une instruction `while`,
- une instruction `do... while`.

**2)** Calculer la moyenne de notes fournies au clavier avec un dialogue de ce type :

```
note 1 : 12
note 2 : 15.25
note 3 : 13.5
note 4 : 8.75
note 5 : -1
moyenne de ces 4 notes : 12.37
```

Le nombre de notes n'est pas connu a priori et l'utilisateur peut en fournir autant qu'il le désire. Pour signaler qu'il a terminé, on convient qu'il fournira une note fictive négative. Celle-ci ne devra naturellement pas être prise en compte dans le calcul de la moyenne.

**3)** Afficher un triangle rempli d'étoiles, s'étendant sur un nombre de lignes fourni en donnée et se présentant comme dans cet exemple :

```
*
**
***
****
*****
```

**4)** Déterminer si un nombre entier fourni en donnée est premier ou non.

**5)** Écrire un programme qui détermine la *n*-ième valeur $u_n$ (*n* étant fourni en donnée) de la « suite de Fibonacci » définie comme suit :

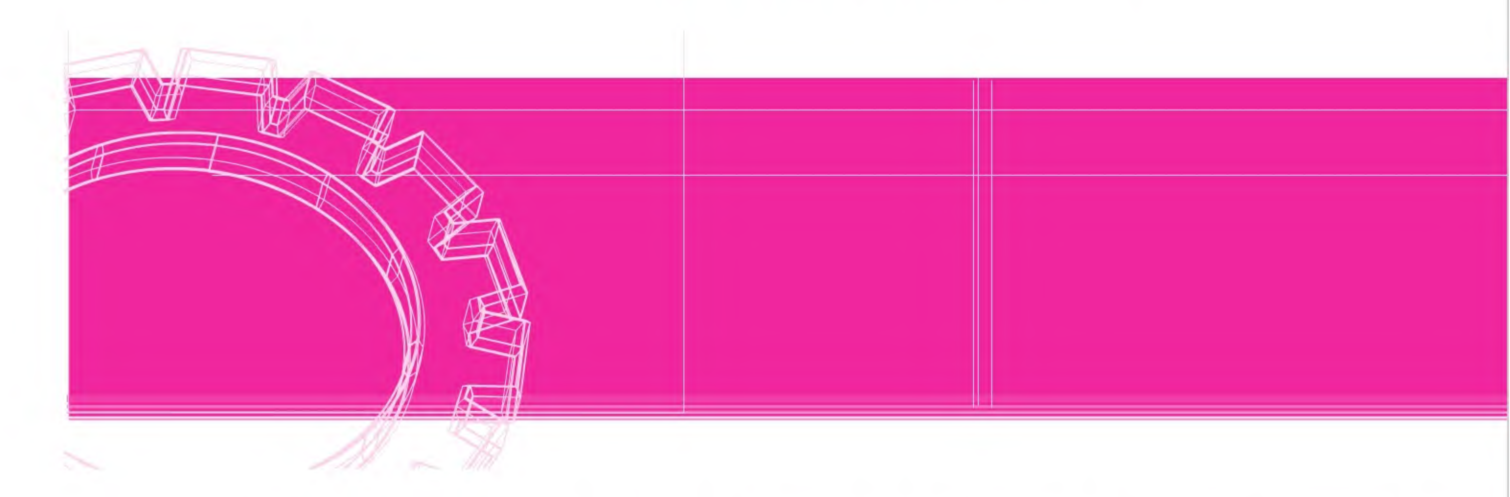
$$u_1 = 1$$
$$u_2 = 1$$
$$u_n = u_{n-1} + u_{n-2} \quad \text{pour } n>2$$

**6)** Écrire un programme qui affiche la table de multiplication des nombres de 1 à 10, sous la forme suivante :

```
      I   1   2   3   4   5   6   7   8   9  10
-----------------------------------------------
   1  I   1   2   3   4   5   6   7   8   9  10
   2  I   2   4   6   8  10  12  14  16  18  20
   3  I   3   6   9  12  15  18  21  24  27  30
   4  I   4   8  12  16  20  24  28  32  36  40
   5  I   5  10  15  20  25  30  35  40  45  50
   6  I   6  12  18  24  30  36  42  48  54  60
   7  I   7  14  21  28  35  42  49  56  63  70
   8  I   8  16  24  32  40  48  56  64  72  80
   9  I   9  18  27  36  45  54  63  72  81  90
  10  I  10  20  30  40  50  60  70  80  90 100
```

# Chapitre 6

# La programmation modulaire et les fonctions



Comme tous les langages, C permet de découper un programme en plusieurs parties nommées souvent « modules ». Cette programmation dite modulaire se justifie pour de multiples raisons :

- Un programme écrit d'un seul tenant devient difficile à comprendre dès qu'il dépasse une ou deux pages de texte. Une écriture modulaire permet de le scinder en plusieurs parties et de regrouper dans le programme principal les instructions en décrivant les enchaînements. Chacune de ces parties peut d'ailleurs, si nécessaire, être décomposée à son tour en modules plus élémentaires ; ce processus de décomposition pouvant être répété autant de fois que nécessaire, comme le préconisent les méthodes de programmation structurée.
- La programmation modulaire permet d'éviter des séquences d'instructions répétitives, et cela d'autant plus que la notion d'argument permet de paramétrer certains modules.
- La programmation modulaire permet le partage d'outils communs qu'il suffit d'avoir écrits et mis au point une seule fois. Cet aspect sera d'autant plus marqué que C autorise effectivement la compilation séparée de tels modules.

# 1 La fonction : la seule sorte de module existant en C

Dans certains langages, on trouve deux sortes de modules, à savoir :

- Les **fonctions**, assez proches de la notion mathématique correspondante. Notamment, une fonction dispose d'arguments (en C, comme dans la plupart des autres langages, une fonction peut ne comporter aucun argument) qui correspondent à des informations qui lui sont transmises et elle fournit un unique résultat scalaire (simple) ; désigné par le nom même de la fonction, ce dernier peut apparaître dans une expression. On dit d'ailleurs que la fonction possède une valeur et qu'un appel de fonction est assimilable à une expression.
- Les **procédures** (terme Pascal) ou sous-programmes (terme Fortran ou Basic) qui élargissent la notion de fonction. La procédure ne possède plus de valeur à proprement parler et son appel ne peut plus apparaître au sein d'une expression. Par contre, elle dispose toujours d'arguments. Parmi ces derniers, certains peuvent, comme pour la fonction, correspondre à des informations qui lui sont transmises. Mais d'autres, contrairement à ce qui se passe pour la fonction, peuvent correspondre à des informations qu'elle produit en retour de son appel. De plus, une procédure peut réaliser une action, par exemple afficher un message (en fait, dans la plupart des langages, la fonction peut quand même réaliser une action, bien que ce ne soit pas là sa vocation).

En C, il n'existe qu'une seule sorte de module, nommé **fonction** (il en ira de même en C++ et en Java, langage dont la syntaxe est proche de celle de C). Ce terme, quelque peu abusif, pourrait laisser croire que les modules du C sont moins généraux que ceux des autres langages. Or il n'en est rien, bien au contraire ! Certes, la fonction pourra y être utilisée comme dans d'autres langages, c'est-à-dire recevoir des arguments et fournir un résultat scalaire qu'on utilisera dans une expression, comme, par exemple, dans :

```
y = sqrt(x)+3 ;
```

Mais, en C, la fonction pourra prendre des aspects différents, pouvant complètement dénaturer l'idée qu'on se fait d'une fonction. Par exemple :

- La valeur d'une fonction pourra très bien ne pas être utilisée ; c'est ce qui se passe fréquemment lorsque vous utilisez `printf` ou `scanf`. Bien entendu, cela n'a d'intérêt que parce que de telles fonctions réalisent une **action** (ce qui, dans d'autres langages, serait réservée aux sous-programmes ou procédures).
- Une fonction pourra ne fournir aucune valeur.
- Une fonction pourra fournir un résultat non scalaire (nous n'en parlerons toutefois que dans le chapitre consacré aux structures).
- Une fonction pourra modifier les valeurs de certains de ses arguments (il vous faudra toutefois attendre d'avoir étudié les pointeurs pour voir par quel mécanisme elle y parviendra).

Ainsi, donc, malgré son nom, en C, la fonction pourra jouer un rôle aussi général que la procédure ou le sous-programme des autres langages.



Par ailleurs, nous verrons qu'en C plusieurs fonctions peuvent partager des informations, autrement que par passage d'arguments. Nous retrouverons la notion classique de « variables globales » (en Basic, toutes les variables sont globales, de sorte qu'on ne le dit pas - en Fortran, ces variables globales sont rangées dans des « COMMON »).

Enfin, l'un des atouts du langage C réside dans la possibilité de **compilation séparée**. Celle-ci permet de découper le programme source en plusieurs parties, chacune de ces parties pouvant comporter une ou plusieurs fonctions. Certains auteurs emploient parfois le mot « module » pour désigner chacune de ces parties (stockées dans un fichier) ; dans ce cas, ce terme de module devient synonyme de fichier source. Cela facilite considérablement le développement et la mise au point de grosses applications. Cette possibilité crée naturellement quelques contraintes supplémentaires, notamment au niveau des variables globales que l'on souhaite partager entre différentes parties du programme source (c'est d'ailleurs ce qui justifiera l'existence de la déclaration extern).

Pour garder une certaine progressivité dans notre exposé, nous supposerons tout d'abord que nous avons affaire à un programme source d'un seul tenant (ce qui ne nécessite donc pas de compilation séparée). Nous présenterons ainsi la structure générale d'une fonction, les notions d'arguments, de variables globales et locales. Ce n'est qu'alors que nous introduirons les possibilités de compilation séparée en montrant quelles sont ses incidences sur les points précédents ; cela nous amènera à parler des différentes « classes d'allocation » des variables.

## 2 Exemple de définition et d'utilisation d'une fonction en C

Nous vous proposons d'examiner tout d'abord un exemple simple de fonction correspondant à l'idée usuelle que l'on se fait d'une fonction, c'est-à-dire recevant des arguments et fournissant une valeur.

*Exemple de définition et d'utilisation d'une fonction*

```
#include <stdio.h>
            /***** le programme principal (fonction main) *****/
main()
{
   float fexple (float, int, int) ; /* déclaration de fonction fexple */
   float x = 1.5 ;
   float y, z ;
   int n = 3, p = 5, q = 10 ;

         /* appel de fexple avec les arguments x, n et p */
   y = fexple (x, n, p) ;
   printf ("valeur de y : %e\n", y) ;
```

*Exemple de définition et d'utilisation d'une fonction (suite)*

```
          /* appel de fexple avec les arguments x+0.5, q et n-1 */
   z = fexple (x+0.5, q, n-1) ;
   printf ("valeur de z : %e\n", z) ;
}

          /*************** la fonction fexple ***************/
float fexple (float x, int b, int c)
{  float val ;          /* déclaration d'une variable "locale" à fexple
   val = x * x + b * x + c ;
   return val ;
}
```

Nous y trouvons tout d'abord, de façon désormais classique, un programme principal formé d'un bloc. Mais, cette fois, à sa suite, apparaît la **définition d'une fonction**. Celle-ci possède une structure voisine de la fonction `main`, à savoir un en-tête et un corps délimité par des accolades ({ et }). Mais l'en-tête est plus élaboré que celui de la fonction `main` puisque, outre le nom de la fonction (`fexple`), on y trouve une liste d'arguments (nom + type), ainsi que le type de la valeur qui sera fournie par la fonction (on la nomme indifféremment « résultat », « valeur de la fonction », « valeur de retour »...) :

```
float          fexple          (float x,        int b,        int c)
  |               |                |               |             |
type de la    nom de la        premier         deuxième      troisième
"valeur       fonction         argument        argument       argument
de retour"                   (type float)     (type int)     (type int)
```

Les noms des arguments n'ont d'importance qu'au sein du corps de la fonction. Ils servent à décrire le travail que devra effectuer la fonction quand on l'appellera en lui fournissant trois valeurs.

Si on s'intéresse au corps de la fonction, on y rencontre tout d'abord une déclaration :

```
float val ;
```

Celle-ci précise que, pour effectuer son travail, notre fonction a besoin d'une variable de type `float` nommée `val`. On dit que `val` est une variable locale à la fonction `fexple`, de même que les variables telles que `n`, `p`, `y`... sont des variables locales à la fonction `main` (mais comme jusqu'ici nous avions affaire à un programme constitué d'une seule fonction, cette distinction n'était pas utile). Un peu plus loin, nous examinerons plus en détail cette notion de variable locale et celle de portée qui s'y attache.

L'instruction suivante de notre fonction `fexple` est une affectation classique (faisant toutefois intervenir les valeurs des arguments `x`, `n` et `p`).

Enfin, l'instruction `return val` précise la valeur que fournira la fonction à la fin de son travail.

En définitive, on peut dire que `fexple` est une fonction telle que `fexple (x, b, c)` fournisse la valeur de l'expression $x^2 + bx + c$. Notez bien l'aspect arbitraire du nom des arguments ; on obtiendrait la même définition de fonction avec, par exemple :

```
float fexple (float z, int coef, int n)
{
   float val ; /* déclaration d'une variable "locale" à fexple */
   val = z * z + coef * z + n ;
   return val ;
}
```

Examinons maintenant la fonction `main`. Vous constatez qu'on y trouve une déclaration :

```
   float fexple (float, int, int) ;
```

Elle sert à prévenir le compilateur que `fexple` est une fonction et elle lui précise le type de ses arguments ainsi que celui de sa valeur de retour. Nous reviendrons plus loin en détail sur le rôle d'une telle déclaration.

Quant à l'utilisation de notre fonction `fexple` au sein de la fonction `main`, elle est classique et comparable à celle d'une fonction prédéfinie telle que `scanf` ou `sqrt`. Ici, nous nous sommes contenté d'appeler notre fonction à deux reprises avec des arguments différents.

# 3 Quelques règles

## 3.1 Arguments muets et arguments effectifs

Les noms des arguments figurant dans l'en-tête de la fonction se nomment des « arguments muets », ou encore « arguments formels » ou « paramètres formels » (de l'anglais : *formal parameter)*. Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.

Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment des « arguments effectifs » (ou encore « paramètres effectifs »). Comme le laisse deviner l'exemple précédent, on peut utiliser n'importe quelle expression comme argument effectif ; au bout du compte, c'est la valeur de cette expression qui sera transmise à la fonction lors de son appel. Notez qu'une telle « liberté » n'aurait aucun sens dans le cas des paramètres formels : il serait impossible d'écrire un en-tête de `fexple` sous la forme `float fexple (float a+b, ...)` pas plus qu'en mathématiques vous ne définiriez une fonction $f$ par $f(x+y)=5$ !

## 3.2 L'instruction return

Voici quelques règles générales concernant cette instruction.

- L'instruction return peut mentionner n'importe quelle expression. Ainsi, nous aurions pu définir la fonction fexple précédente d'une manière plus simple :

```
float fexple (float x, int b, int c)
{
   return (x * x + b * x + c) ;
}
```

- L'instruction return peut apparaître à plusieurs reprises dans une fonction, comme dans cet autre exemple :

```
double absom (double u, double v)
{
    double s ;
    s = a + b ;
    if (s>0)     return (s) ;
         else    return (-s)
}
```

  Notez bien que non seulement l'instruction return définit la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée (n'oubliez pas qu'en C tous les modules sont des fonctions, y compris le programme principal). Nous verrons qu'une fonction peut ne fournir aucune valeur : elle peut alors disposer de une ou plusieurs instructions return **sans expression**, interrompant simplement l'exécution de la fonction ; mais elle peut aussi dans ce cas ne comporter aucune instruction return, le retour étant alors mis en place automatiquement par le compilateur à la fin de la fonction.

- Si le type de l'expression figurant dans return est différent du type du résultat tel qu'il a été déclaré dans l'en-tête, le compilateur mettra automatiquement en place des instructions de conversion.

  Il est toujours possible de ne pas utiliser le résultat d'une fonction, même si elle en produit un. C'est d'ailleurs ce que nous avons fait fréquemment avec printf ou scanf. Bien entendu, cela n'a d'intérêt que si la fonction fait autre chose que de calculer un résultat. En revanche, il est interdit d'utiliser la valeur d'une fonction ne fournissant pas de résultat (si certains compilateurs l'acceptent, vous obtiendrez, lors de l'exécution, une valeur aléatoire !).

## 3.3 Cas des fonctions sans valeur de retour ou sans arguments

Quand une fonction ne renvoie pas de résultat, on le précise, à la fois dans l'en-tête et dans sa déclaration, à l'aide du mot-clé `void`. Par exemple, voici l'en-tête d'une fonction recevant un argument de type `int` et ne fournissant aucune valeur :

```
void sansval (int n)
```

et voici quelle serait sa déclaration :

```
void sansval (int) ;
```

Naturellement, la définition d'une telle fonction ne doit, en principe, contenir aucune instruction `return`. Certains compilateurs ne détecteront toutefois pas l'erreur.

Quand une fonction ne reçoit aucun argument, on place le mot-clé `void` (le même que précédemment, mais avec une signification différente !) à la place de la liste d'arguments (attention, en C++, la règle sera différente : on se contentera de ne rien mentionner dans la liste d'arguments). Voici l'en-tête d'une fonction ne recevant aucun argument et renvoyant une valeur de type `float` (il pourrait s'agir, par exemple, d'une fonction fournissant un nombre aléatoire !) :

```
float tirage (void)
```

Sa déclaration serait très voisine (elle ne diffère que par la présence du point-virgule !) :

```
float tirage (void) ;
```

Enfin, rien n'empêche de réaliser une fonction ne possédant ni arguments ni valeur de retour. Dans ce cas, son en-tête sera de la forme :

```
void message (void)
```

et sa déclaration sera :

```
void message (void) ;
```

**Remarque**

En toute rigueur, la fonction `main` est une fonction sans argument et sans valeur de retour. Elle devrait donc avoir pour en-tête « `void main (void)` ». Certains compilateurs fournissent d'ailleurs un message d'avertissement (« warning ») lorsque vous vous contentez de l'en-tête usuel `main`.

Voici un exemple illustrant deux des situations évoquées. Nous y définissons une fonction `affiche_carres` qui affiche les carrés des nombres entiers compris entre deux limites fournies en arguments et une fonction `erreur` qui se contente d'afficher un message d'erreur (il s'agit de notre premier exemple de programme source contenant plus de deux fonctions).

```
#include <stdio.h>
main()
{  void affiche_carres (int, int) ; /* prototype de affiche_carres */
   void erreur (void) ;             /* prototype de erreur */
   int debut = 5, fin = 10 ;
      .....
   affiche_carres (debut, fin) ;
      .....
   if (...) erreur () ;
}
void affiche_carres (int d, int f)
{  int i ;
   for (i=d ; i<=f ; i++)
      printf ("%d a pour carré %d\n", i, i*i) ;
}
void erreur (void)
{  printf ("*** erreur ***\n") ; }
```

## 3.4 Les anciennes formes de l'en-tête des fonctions

Dans la première version du langage C, telle qu'elle a été définie par Kernighan et Ritchie, avant la normalisation par le comité ANSI, l'en-tête d'une fonction s'écrivait différemment de ce que nous avons vu ici. Par exemple, l'en-tête de notre fonction `fexple` aurait été :

```
float fexple (x, b, c)
float x ;
int b, c ;
```

La norme ANSI autorise les deux formes, lesquelles sont actuellement acceptées par la plupart des compilateurs. Toutefois, seule la forme moderne, c'est-à-dire celle que nous avons présentée précédemment, sera autorisée par C++.

**Remarque**

L'habitude veut que les en-têtes écrits sous l'ancienne forme le soient sur plusieurs lignes comme dans notre exemple. Mais rien ne nous empêcherait de l'écrire sous cette forme :

```
float fexple (x, b, c) float x ; int b, c ;
```

# 4 Les fonctions et leurs déclarations

## 4.1 Les différentes façons de déclarer (ou de ne pas déclarer) une fonction

Dans notre exemple du paragraphe 2, nous avions fourni la définition de la fonction `fexple` **après** celle de la fonction `main`. Mais nous aurions pu tout aussi bien faire l'inverse :

```
float fexple (float x, int b, int c)
{
    ....
}



main()
{
   float fexple (float, int, int) ; /* déclaration de la fonc. fexple */
       .....
   y = fexple (x, n, p) ;
       .....
}
```

En toute rigueur, dans ce cas, la déclaration de la fonction `fexple` (ici, dans `main`) est facultative, car, lorsqu'il traduit la fonction `main`, le compilateur connaît déjà la fonction `fexple`. Néanmoins, nous vous déconseillons d'omettre la déclaration de `fexple` dans ce cas ; en effet, il est tout à fait possible qu'ultérieurement vous soyez amené à modifier votre programme source ou même à l'éclater en plusieurs fichiers source comme l'autorisent les possibilités de compilation séparée du langage C.

Par ailleurs, le langage C (mais pas le C++) vous permet d'effectuer des déclarations partielles en ne mentionnant pas le type des arguments ; ainsi, dans notre exemple du paragraphe 2, nous pourrions déclarer `fexple` de cette façon dans la fonction `main` :

```
float fexple () ;
```

Qui plus est, C vous autorise à ne pas déclarer du tout une fonction qui renvoie une valeur de type `int` (là encore, ce sera interdit en C++ ainsi qu'en C99).

Nous ne saurions trop vous conseiller d'éviter de telles possibilités. Toutefois, sachez que vous risquez d'employer la dernière sans y prendre garde. En effet, toute fonction que vous utiliserez sans l'avoir déclarée sera considérée par le compilateur comme ayant des arguments quelconques et fournissant un résultat de type `int`. Les conséquences en seront différentes suivant que ladite fonction est ou non fournie dans le même fichier source. Dans le premier cas, on obtiendra bien une erreur de compilation ; dans le second, en revanche, les conséquences n'apparaîtront (de manière plus ou moins voilée) que lors de l'exécution !

La déclaration complète d'une fonction porte le nom de **prototype**. Il est possible, dans un prototype, de faire figurer des noms d'arguments, lesquels sont alors totalement arbitraires ; cette possibilité a pour seul intérêt de pouvoir écrire des prototypes qui sont identiques à l'en-tête de la fonction (au point-virgule près), ce qui peut en faciliter la création automatique. Dans notre exemple du paragraphe 2, notre fonction `fexple` aurait pu être **déclarée** ainsi :

```
float fexple (float x, int b, int c) ;
```

## 4.2 Où placer la déclaration d'une fonction

La tendance la plus naturelle consiste à placer la déclaration d'une fonction à l'intérieur des déclarations de toute fonction l'utilisant ; c'est ce que nous avons fait jusqu'ici. Et, de surcroît, dans tous nos exemples précédents, la fonction utilisatrice était la fonction `main` elle-même ! Dans ces conditions, nous avions affaire à une déclaration locale dont la portée était limitée à la fonction où elle apparaissait.

Mais il est également possible d'utiliser des déclarations globales, en les faisant apparaître **avant la définition de la première fonction**. Par exemple, avec :

```
float fexple (float, int, int) ;
main()
{  .....
}
void f1 (...)
{  .....
}
```

la déclaration de `fexple` est connue à la fois de `main` et de `f1`.

## 4.3 À quoi sert la déclaration d'une fonction

Nous avons vu que la déclaration d'une fonction est plus ou moins obligatoire et qu'elle peut être plus ou moins détaillée. Malgré tout, nous vous avons recommandé d'employer toujours la forme la plus complète possible qu'on nomme prototype. Dans ce cas, un tel prototype peut être utilisé par le compilateur, et cela de deux façons complètement différentes.

**a)** Si la définition de la fonction se trouve dans le même fichier source (que ce soit avant ou après la déclaration), il s'assure que les arguments muets ont bien le type défini dans le prototype. Dans le cas contraire, il signale une erreur.

**b)** Lorsqu'il rencontre un appel de la fonction, il met en place d'éventuelles conversions des valeurs des arguments effectifs dans le type indiqué dans le prototype. Par exemple, avec notre fonction `fexple` du paragraphe 2, un appel tel que :

```
fexple (n+1, 2*x, p)
```

sera traduit par :

- l'évaluation de la valeur de l'expression `n+1` (en `int`) et sa conversion en `float`,
- l'évaluation de la valeur de l'expression `2*x` (en `float`) et sa conversion en `int` ; il y a donc dans ce dernier cas une conversion dégradante.

**Remarques**

Rappelons que, lorsque le compilateur ne connaît pas le type des arguments d'une fonction, il utilise des règles de conversions systématiques : `char` *et* `short -> int` et `float -> double`. La fonction `printf` est précisément dans ce cas.

Compte tenu de la remarque précédente, seule une fonction déclarée avec un prototype pourra recevoir un argument de type `float`, `char` ou `short`.

# 5 Retour sur les fichiers en-tête

Nous avons déjà dit qu'il existe un certain nombre de fichiers d'extension `.h`, correspondant chacun à une classe de fonctions. On y trouve, entre autres choses, les prototypes de ces fonctions.

Ce point se révèle fort utile :

- d'une part pour effectuer des contrôles sur le nombre et le type des arguments mentionnés dans les appels de ces fonctions,
- d'autre part pour forcer d'éventuelles conversions auxquelles on risque de ne pas penser.

À titre d'illustration de ce dernier aspect, supposez que vous ayez écrit ces instructions :

```
float x, y ;
    .....
y = sqrt (x) ;
    .....
```

sans les faire précéder d'une quelconque directive `#include`.

Elles produiraient alors des **résultats faux**. En effet, il se trouve que la fonction `sqrt` s'attend à recevoir un argument de type `double` (ce qui sera le cas ici, compte tenu des conversions implicites), et elle fournit un résultat de type `double`. Or, lors de la traduction de votre programme, le compilateur ne le sait pas. Il attribue donc d'office à `sqrt` le type `int` et il met en place une conversion de la valeur de retour (laquelle sera en fait de type `double`) en `int`. On se trouve en présence des conséquences habituelles d'une mauvaise interprétation de type.

Un premier remède consiste à placer dans votre module la déclaration :

```
double sqrt(double) ;
```

mais encore faut-il que vous connaissiez de façon certaine le type de cette fonction.

Une meilleure solution consiste à placer, en début de votre programme, la directive :

```
#include <math.h>
```

laquelle incorporera automatiquement le prototype approprié (entre autres choses).

## 6 En C, les arguments sont transmis par valeur

Nous avons déjà eu l'occasion de dire qu'en C les arguments d'une fonction étaient transmis par valeur. Cependant, dans les exemples que nous avons rencontrés dans ce chapitre, les conséquences et les limitations de ce mode de transmission n'apparaissaient guère. Or voyez cet exemple :

*Conséquences de la transmission par valeur des arguments*

```
#include <stdio.h>
main()
{  void echange (int a, int b) ;
   int n=10, p=20 ;
   printf ("avant appel    : %d %d\n", n, p) ;
   echange (n, p) ;
   printf ("après appel    : %d %d", n, p)
}
void echange (int a, int b)
{
   int c ;
   printf ("début echange : %d %d\n", a, b) ;
   c = a ;
   a = b ;
   b = c ;
   printf ("fin echange    : %d %d\n", a, b) ;
}
```

```
avant appel   : 10 20
début echange : 10 20
fin echange   : 20 10
après appel   : 10 20
```

La fonction `echange` reçoit deux valeurs correspondant à ses deux arguments muets `a` et `b`. Elle effectue un échange de ces deux valeurs. Mais, lorsque l'on est revenu dans le programme principal, aucune trace de cet échange ne subsiste sur les arguments effectifs `n` et `p`.

En effet, lors de l'appel de `echange`, il y a eu transmission de la valeur des expressions `n` et `p`. On peut dire que ces valeurs ont été recopiées localement dans la fonction `echange` dans



des emplacements nommés `a` et `b`. C'est effectivement sur ces copies qu'a travaillé la fonction `echange`, de sorte que les valeurs des variables `n` et `p` n'ont, quant à elles, pas été modifiées. C'est ce qui explique le résultat constaté.

Ce mode de transmission semble donc interdire a priori qu'une fonction produise une ou plusieurs valeurs en retour, autres que celle de la fonction elle-même.

Or, il ne faut pas oublier qu'en C tous les modules doivent être écrits sous forme de fonction. Autrement dit, ce simple problème d'échange des valeurs de deux variables doit pouvoir se résoudre à l'aide d'une fonction.

Nous verrons que ce problème possède plusieurs solutions, à savoir :

- Transmettre en argument la valeur de l'adresse d'une variable. La fonction pourra éventuellement agir sur le contenu de cette adresse. C'est précisément ce que nous faisons lorsque nous utilisons la fonction `scanf`. Nous examinerons cette technique en détail dans le chapitre consacré aux pointeurs.
- Utiliser des variables globales, comme nous le verrons dans le prochain paragraphe ; cette deuxième solution devra toutefois être réservée à des cas exceptionnels, compte tenu des risques qu'elle présente (effets de bords).

**Remarques**

C'est bien parce que la transmission des arguments se fait « par valeur » que les arguments effectifs peuvent prendre la forme d'une expression quelconque. Dans les langages où le seul mode de transmission est celui « par adresse », les arguments effectifs ne peuvent être que l'équivalent d'une *lvalue*.

La norme n'impose aucun ordre pour l'évaluation des différents arguments d'une fonction lors de son appel. En général, ceci est de peu d'importance, excepté dans une situation (fortement déconseillée !) telle que :

```
int i = 10 ;
    ...
f (i++, i) ;    /* on peut calculer i++ avant i --> f (10, 11) */
                /*                     ou après i --> f (10, 10) */
```

## 7 Les variables globales

Nous avons vu comment échanger des informations entre différentes fonctions grâce à la transmission d'arguments et à la récupération d'une valeur de retour.

En fait, en C, plusieurs fonctions (dont, bien entendu le programme principal `main`) peuvent partager des variables communes qu'on qualifie alors de **globales**.

**Remarque**

La norme ANSI ne parle pas de variables globales, mais de variables externes. Le terme « global » illustre plutôt le partage entre plusieurs fonctions tandis que le terme « externe » illustre plutôt le partage entre plusieurs fichiers source. En C, une variable globale est partagée par plusieurs fonctions ; elle peut être (mais elle n'est pas obligatoirement) partagée entre plusieurs fichiers source.

## 7.1 Exemple d'utilisation de variables globales

Voyez cet exemple de programme.

*Exemple d'utilisation d'une variable globale*

```
#include <stdio.h>
int i ;
main()
{  void optimist (void) ;
   for (i=1 ; i<=5 ; i++)
       optimist() ;
}
void optimist(void)
{   printf ("il fait beau %d fois\n", i) ;
}
```

```
il fait beau 1 fois
il fait beau 2 fois
il fait beau 3 fois
il fait beau 4 fois
il fait beau 5 fois
```

La variable `i` a été déclarée en dehors de la fonction `main`. Elle est alors connue de toutes les fonctions qui seront compilées par la suite au sein du même programme source. Ainsi, ici, le programme principal affecte à `i` des valeurs qui se trouvent utilisées par la fonction `optimist`.

Notez qu'ici la fonction `optmist` se contente d'utiliser la valeur de `i` mais rien ne l'empêche de la modifier. C'est précisément ce genre de remarque qui doit vous inciter à n'utiliser les variables globales que dans des cas limités. En effet, toute variable globale peut être modifiée insidieusement par n'importe quelle fonction. Lorsque vous aurez à écrire des fonctions susceptibles de modifier la valeur de certaines variables, il sera beaucoup plus judicieux de prévoir d'en transmettre l'adresse en argument (comme vous apprendrez à le faire dans le prochain chapitre). En effet, dans ce cas, l'appel de la fonction montrera explicitement quelle est la variable qui risque d'être modifiée et, de plus, ce sera la seule qui pourra l'être.

## 7.2 La portée des variables globales

Les variables globales ne sont connues du compilateur que dans la partie du programme source suivant leur déclaration. On dit que leur **portée** (ou encore leur **espace de validité**) est limitée à la partie du programme source qui suit leur déclaration (n'oubliez pas que, pour l'instant, nous nous limitons au cas où l'ensemble du programme est compilé en une seule fois).

Ainsi, voyez, par exemple, ces instructions :

```
main()
{
   ....
}
int n ;
float x ;
fct1 (...)
{
   ....
}
fct2 (...)
{
   ....
}
```

Les variables `n` et `x` sont accessibles aux fonctions `fct1` et `fct2`, mais pas au programme principal. En pratique, bien qu'il soit possible effectivement de déclarer des variables globales à n'importe quel endroit du programme source qui soit extérieur aux fonctions, on procédera rarement ainsi. En effet, pour d'évidentes raisons de lisibilité, on préférera regrouper les déclarations de toutes les variables globales au début du programme source.

### 7.3 La classe d'allocation des variables globales

D'une manière générale, les variables globales existent pendant toute l'exécution du programme dans lequel elles apparaissent. Leurs emplacements en mémoire sont parfaitement définis lors de l'édition de liens. On traduit cela en disant qu'elles font partie de la **classe d'allocation statique**.

De plus, ces variables se voient **initialisées à zéro**, avant le début de l'exécution du programme, sauf, bien sûr, si vous leur attribuez explicitement une valeur initiale au moment de leur déclaration.

## 8 Les variables locales

À l'exception de l'exemple du paragraphe précédent, les variables que nous avions rencontrées jusqu'ici n'étaient pas des variables globales. Plus précisément, elles étaient définies au sein d'une fonction (qui pouvait être `main`). De telles variables sont dites **locales** à la fonction dans laquelle elles sont déclarées.

## 8.1 La portée des variables locales

Les variables locales ne sont connues qu'à l'intérieur de la fonction où elles sont déclarées. **Leur portée est donc limitée à cette fonction.**

Les variables locales n'ont aucun lien avec des variables globales de même nom ou avec d'autres variables locales à d'autres fonctions.

Voyez cet exemple :

```
int n ;
main()
{
   int p ;
   ....
}
fct1 ()
{
   int p ;
   int n ;
}
```

La variable `p` de `main` n'a aucun rapport avec la variable `p` de `fct1`. De même, la variable `n` de `fct1` n'a aucun rapport avec la variable globale `n`. Notez qu'il est alors impossible, dans la fonction `fct1`, d'utiliser cette variable globale `n`.

## 8.2 Les variables locales automatiques

Par défaut, les variables locales ont une durée de vie limitée à celle d'**une exécution** de la fonction dans laquelle elles figurent.

Plus précisément, leurs emplacements ne sont pas définis de manière permanente comme ceux des variables globales. Un nouvel espace mémoire leur est alloué à chaque entrée dans la fonction et libéré à chaque sortie. Il sera donc généralement différent d'un appel au suivant.

On traduit cela en disant que la **classe d'allocation** de ces variables est **automatique**. Nous aurons l'occasion de revenir plus en détail sur cette gestion dynamique de la mémoire. Pour l'instant, il est important de noter que la conséquence immédiate de ce mode d'allocation est que les valeurs des variables locales ne sont pas conservées d'un appel au suivant (on dit aussi qu'elles ne sont pas « rémanentes »). Nous reviendrons un peu plus loin (paragraphe 11.2) sur les éventuelles initialisations de telles variables.

D'autre part, les valeurs transmises en arguments à une fonction sont traitées de la même manière que les variables locales. Leur durée de vie correspond également à celle de la fonction.

## 8.3 Les variables locales statiques

Il est toutefois possible de demander d'attribuer un emplacement permanent à une variable locale et qu'ainsi sa valeur se conserve d'un appel au suivant. Il suffit pour cela de la déclarer à l'aide du mot-clé **`static`** (le mot `static` employé sans indication de type est équivalent à `static int`).

En voici un exemple :

*Exemple d'utilisation de variable locale statique*

```
#include <stdio.h>
main()
{  void fct(void) ;
   int n ;
   for ( n=1 ; n<=5 ; n++)
      fct() ;
}
void fct(void)
{  static int i ;
   i++ ;
   printf ("appel numéro : %d\n", i) ;
}
```

```
appel numéro : 1
appel numéro : 2
appel numéro : 3
appel numéro : 4
appel numéro : 5
```

La variable locale `i` a été déclarée de classe « statique ». On constate bien que sa valeur progresse de un à chaque appel. De plus, on note qu'au premier appel sa valeur est nulle. En effet, comme pour les variables globales (lesquelles sont aussi de classe statique) : **les variables locales de classe statique sont, par défaut, initialisées à zéro**.

Prenez garde à ne pas confondre une variable locale de classe statique avec une variable globale. En effet, la portée d'une telle variable reste toujours limitée à la fonction dans laquelle elle est définie. Ainsi, dans notre exemple, nous pourrions définir une variable globale nommée `i` qui n'aurait alors aucun rapport avec la variable `i` de `fct`.

## 8.4 Le cas des fonctions récursives

Le langage C autorise la récursivité des appels de fonctions. Celle-ci peut prendre deux aspects :

- récursivité directe : une fonction comporte, dans sa définition, au moins un appel à elle-même,
- récursivité croisée : l'appel d'une fonction entraîne celui d'une autre fonction qui, à son tour, appelle la fonction initiale (le cycle pouvant d'ailleurs faire intervenir plus de deux fonctions).

Voici un exemple fort classique (d'ailleurs inefficace sur le plan du temps d'exécution) d'une fonction calculant une factorielle de manière récursive :

*Fonction récursive de calcul de factorielle*

```
long fac (int n)
{
   if (n>1) return (fac(n-1)*n) ;
        else return(1) ;
}
```

Il faut bien voir qu'alors chaque appel de `fac` entraîne une allocation d'espace pour les variables locales et pour son argument `n` (apparemment, `fct` ne comporte aucune variable locale ; en réalité, il lui faut prévoir un emplacement destiné à recevoir sa valeur de retour). Or chaque nouvel appel de `fac`,à l'intérieur de `fac`, provoque une telle allocation, sans que les emplacements précédents soient libérés.

Il y a donc un empilement des espaces alloués aux variables locales, parallèlement à un empilement des appels de la fonction. Ce n'est que lors de l'exécution de la première instruction `return` que l'on commencera à « dépiler » les appels et les emplacements et donc à libérer de l'espace mémoire.

# 9 La compilation séparée et ses conséquences

Si le langage C est effectivement un langage que l'on peut qualifier d'opérationnel, c'est en partie grâce à ses possibilités dites de **compilation séparée**. En C, en effet, il est possible de compiler séparément plusieurs programmes (fichiers) source et de rassembler les modules objet correspondants au moment de l'édition de liens. D'ailleurs, dans certains environnements de programmation, la notion de *projet* permet de gérer la multiplicité des fichiers (source et modules objet) pouvant intervenir dans la création d'un programme exécutable. Cette notion de projet fait intervenir précisément les fichiers à considérer ; généralement, il est possible de demander de créer le programme exécutable, en ne recompilant que les sources ayant subi une modification depuis leur dernière compilation.

Indépendamment de ces aspects techniques liés à l'environnement de programmation considéré, les possibilités de compilation séparée ont une incidence importante au niveau de la portée des variables globales. C'est cet aspect que nous nous proposons d'étudier maintenant. Dans le



paragraphe suivant, nous serons alors en mesure de faire le point sur les différentes classes d'allocation des variables.

Notez que, à partir du moment où l'on parle de compilation séparée, il existe au moins (ou il a existé) deux programmes source ; dans la suite, nous supposerons qu'ils figurent dans des fichiers, de sorte que nous parlerons toujours de fichier source.

Pour l'instant, voyons l'incidence de cette compilation séparée sur la portée des variables globales.

## 9.1 La portée d'une variable globale - la déclaration extern

A priori, la portée d'une variable globale semble limitée au fichier source dans lequel elle a été définie. Ainsi, supposez que l'on compile séparément ces deux fichiers source :

```
source 1                  source 2
 int x ;                    fct2()
 main()                     {
 {                              .....
    .....                   }
 }                          fct3()
 fct1()                     {
 {                              .....
    .....                   }
 }
```

Il ne semble pas possible, dans les fonctions `fct2` et `fct3` de faire référence à la variable globale `x` déclarée dans le premier fichier source (alors qu'aucun problème ne se poserait si l'on réunissait ces deux fichiers source en un seul, du moins si l'on prenait soin de placer les instructions du second fichier à la suite de celles du premier).

En fait, le langage C prévoit une déclaration permettant de spécifier qu'une variable globale a déjà été définie dans un autre fichier source. Celle-ci se fait à l'aide du mot-clé `extern`. Ainsi, en faisant précéder notre second fichier source de la déclaration :

```
extern int x ;
```

il devient possible de mentionner la variable globale `x` (déclarée dans le premier fichier source) dans les fonctions `fct2` et `fct3`.

**Remarques**

Cette déclaration `extern` n'effectue **pas de réservation d'emplacement de variable**. Elle ne fait que préciser que la variable globale `x` est définie par ailleurs et elle en précise le type.

Nous n'avons considéré ici que la façon la plus usuelle de gérer des variables globales (celle-ci est utilisable avec tous les compilateurs, qu'ils soient d'avant ou d'après la norme). La norme prévoit d'autres possibilités, au demeurant fort peu répandues et, de surcroît, peu logiques (doubles déclarations, mot `extern` utilisé même pour la réservation d'un emplacement, définitions potentielles).

## 9.2 Les variables globales et l'édition de liens

Supposons que nous ayons compilé les deux fichiers source précédents et voyons d'un peu plus près comment l'éditeur de liens est en mesure de rassembler correctement les deux modules objet ainsi obtenus. En particulier, examinons comment il peut faire correspondre au symbole `x` du second fichier source l'adresse effective de la variable `x` définie dans le premier.

D'une part, après compilation du premier fichier source, on trouve, dans le module objet correspondant, une indication associant le symbole `x` et son adresse dans le module objet. Autrement dit, contrairement à ce qui se produit pour les variables locales, pour lesquelles ne subsiste aucune trace du nom après compilation, le nom des variables globales continue à exister au niveau des modules objet. On retrouve là un mécanisme analogue à ce qui se passe pour les noms de fonctions, lesquels doivent bien subsister pour que l'éditeur de liens soit en mesure de retrouver les modules objet correspondants.

D'autre part, après compilation du second fichier source, on trouve dans le module objet correspondant, une indication mentionnant qu'une certaine variable de nom `x` provient de l'extérieur et qu'il faudra en fournir l'adresse effective.

Ce sera effectivement le rôle de l'éditeur de liens que de retrouver dans le premier module objet l'adresse effective de la variable `x` et de la reporter dans le second module objet.

Ce mécanisme montre que s'il est possible, par mégarde, de réserver des variables globales de même nom dans deux fichiers source différents, il sera, par contre, en général, impossible d'effectuer correctement l'édition de liens des modules objet correspondants (certains éditeurs de liens peuvent ne pas détecter cette anomalie). En effet, dans un tel cas, l'éditeur de liens se trouvera en présence de deux adresses différentes pour un même identificateur, ce qui est illogique.

## 9.3 Les variables globales cachées - la déclaration static

Il est possible de « cacher » une variable globale, c'est-à-dire de la rendre inaccessible à l'extérieur du fichier source où elle a été définie (on dit aussi « rendre confidentielle » au lieu de « cacher » ; on parle alors de « variables globales confidentielles »). Il suffit pour cela d'utiliser la déclaration **`static`** comme dans cet exemple :

```
static int a ;
main()
{
   .....
}
fct()
{
   .....
}
```



Sans la déclaration `static`, `a` serait une variable globale ordinaire. Par contre, cette déclaration demande qu'aucune trace de `a` ne subsiste dans le module objet résultant de ce fichier source. Il sera donc impossible d'y faire référence depuis une autre source par `extern`. Mieux, si une autre variable globale apparaît dans un autre fichier source, elle sera acceptée à l'édition de liens puisqu'elle ne pourra pas interférer avec celle du premier source.

Cette possibilité de cacher des variables globales peut s'avérer précieuse lorsque vous êtes amené à développer un ensemble de fonctions d'intérêt général qui doivent partager des variables globales. En effet, elle permet à l'utilisateur éventuel de ces fonctions de ne pas avoir à se soucier des noms de ces variables globales puisqu'il ne risque pas alors d'y accéder par mégarde. Cela généralise en quelque sorte à tout un fichier source la notion de variable locale telle qu'elle était définie pour les fonctions. Ce sont d'ailleurs de telles possibilités qui permettent de développer des logiciels en C, en utilisant une philosophie orientée objet.

# 10 Les différents types de variables, leur portée et leur classe d'allocation

Nous avons déjà vu différentes choses concernant les classes d'allocation des variables et leur portée. Ici, nous nous proposons de faire le point après avoir introduit quelques informations supplémentaires.

## 10.1 La portée des variables

On peut classer les variables en quatre catégories en fonction de leur portée (ou espace de validité). Nous avons déjà rencontré les trois premières que sont : les variables globales, les variables globales cachées et les variables locales à une fonction. En outre, il est possible de définir des **variables locales à un bloc**. Elles se déclarent en début d'un bloc de la même façon qu'en début d'une fonction. Dans ce cas, la portée de telles variables est limitée au bloc en question. Leur usage est, en pratique, peu répandu.

## 10.2 Les classes d'allocation des variables

Il est également possible de classer les variables en trois catégories en fonction de leur classe d'allocation. Là encore, nous avons déjà rencontré les deux premières, à savoir :

- **la classe statique** : elle renferme non seulement les variables globales (quelles qu'elles soient), mais aussi les variables locales faisant l'objet d'une déclaration `static`. Les emplacements mémoire correspondants sont alloués une fois pour toutes au moment de l'édition de liens,

- **la classe automatique** : par défaut, les variables locales entrent dans cette catégorie. Les emplacements mémoire correspondants sont alloués à chaque entrée dans la fonction où sont définies ces variables et ils sont libérés à chaque sortie.

En toute rigueur, il existe une classe un peu particulière, à savoir la **classe registre :** toute variable entrant a priori dans la classe automatique peut être déclarée explicitement avec le qualificatif **`register`**. Celui-ci demande au compilateur d'utiliser, dans la mesure du possible, un registre pour y ranger la variable ; cela peut amener quelques gains de temps d'exécution. Bien entendu, cette possibilité ne peut s'appliquer qu'aux variables scalaires.

**Remarque**

**Le cas des fonctions.** La fonction est considérée par le langage C comme un objet global. C'est ce qui permet d'ailleurs à l'éditeur de liens d'effectuer correctement son travail. Il faut noter toutefois qu'il n'est pas nécessaire d'utiliser une déclaration `extern` pour les fonctions définies dans un fichier source différent de celui où elles sont appelées (mais le faire ne constitue pas une erreur).

En tant qu'objet global, la fonction peut voir sa portée limitée au fichier source dans lequel elle est définie à l'aide d'une déclaration `static` comme dans :

```
static int fct (...)
```

## 10.3 Tableau récapitulatif

Voici un tableau récapitulant la portée et la classe d'allocation des différentes variables suivant la nature de leur déclaration (la colonne « Type » donne le nom qu'on attribue usuellement à de telles variables).

*Type, portée et classe d'allocation des variables*

<table>
<tr><th>Type de variable</th><th>Déclaration</th><th>Portée</th><th>Classe d'allocation</th></tr>
<tr><td>Globale</td><td>en dehors de toute fonction</td><td>• la partie du fichier source suivant sa déclaration,<br>• n'importe quel fichier source, avec <code>extern</code>.</td><td rowspan="3">Statique</td></tr>
<tr><td>Globale cachée</td><td>en dehors de toute fonction, avec l'attribut <code>static</code></td><td>uniquement la partie du fichier source suivant sa déclaration</td></tr>
<tr><td>Locale<br>« rémanente »</td><td>au début d'une fonction, avec l'attribut <code>static</code></td><td>la fonction</td></tr>
<tr><td>Locale à une fonction</td><td>au début d'une fonction</td><td>la fonction</td><td rowspan="2">Automatique</td></tr>
<tr><td>Locale à un bloc</td><td>au début d'un bloc</td><td>le bloc</td></tr>
</table>

# 11 Initialisation des variables

Nous avons vu qu'il était possible d'initialiser explicitement une variable lors de sa déclaration. Ici, nous allons faire le point sur ces possibilités, lesquelles dépendent en fait de la classe d'allocation de la variable concernée.

## 11.1 Les variables de classe statique

Ces variables sont permanentes. Elles sont initialisées une seule fois avant le début de l'exécution du programme.

Elles peuvent être initialisées explicitement lors de leur déclaration. Nous verrons que cela s'applique également aux tableaux ou aux structures. Bien entendu, les valeurs servant à ces initialisations ne pourront être que des constantes ou des expressions constantes (c'est-à-dire calculables par le compilateur). N'oubliez pas que les constantes symboliques définies par `const` ne sont pas considérées comme des expressions constantes.

En l'absence d'initialisation explicite, ces variables seront initialisées à zéro.

## 11.2 Les variables de classe automatique

Ces variables ne sont pas initialisées par défaut. En revanche, comme les variables de classe statique, elles peuvent être initialisées explicitement lors de leur déclaration.

Dans ce cas, lorsqu'il s'agit de variables scalaires (ce qui est le cas de toutes celles rencontrées jusqu'ici, mais ne sera pas le cas des tableaux ou des structures), la norme vous autorise à utiliser comme valeur initiale non seulement une valeur constante, mais également n'importe quelle expression (dans la mesure où sa valeur est définie au moment de l'entrée dans la fonction correspondante, laquelle, ne l'oubliez pas, peut être la fonction `main`).

En voici un cas d'école :

*Initialisation de variables de classe automatique*

```
#include <stdio.h>
int n ;
main()
{ void fct (int r) ;
  int p ;
  for (p=1 ; p<=5 ; p++)
    { n = 2*p ;
      fct(p) ;
    }
}
void fct(int r)
{
  int q=n, s=r*n ;
  printf ("%d %d %d\n", r,q,s) ;
}
```

N'oubliez pas que ces variables automatiques se trouvent alors initialisées à chaque appel de la fonction dans laquelle elles sont définies.

# 12 Les arguments variables en nombre

Dans tous nos précédents exemples, le nombre d'arguments fournis au cours de l'appel d'une fonction était prévu lors de l'écriture de cette fonction.

Or, dans certaines circonstances, on peut souhaiter réaliser une fonction capable de recevoir un nombre d'arguments susceptible de varier d'un appel à un autre. C'est d'ailleurs ce que nous faisons (sans même y penser) lorsque nous utilisons des fonctions comme `printf` ou `scanf` (en dehors du premier argument, qui représente le format, les autres sont en nombre quelconque)

Le langage C permet de résoudre ce problème à l'aide des fonctions particulières `va_start` et `va_arg`. La seule contrainte à respecter est que la fonction doit posséder certains arguments fixes (c'est-à-dire toujours présents), leur nombre ne pouvant être inférieur à un. En effet, comme nous allons le voir, c'est le dernier argument fixe qui permet, en quelque sorte, d'initialiser le parcours de la liste d'arguments.

## 12.1 Premier exemple

Voici un premier exemple de fonction à arguments variables : les deux premiers arguments sont fixes, l'un étant de type `int`, l'autre de type `char`. Les arguments suivants, de type `int`, sont en nombre quelconque et l'on a supposé que le dernier d'entre eux était `-1`. Cette dernière valeur sert donc, en quelque sorte, de sentinelle. Par souci de simplification, nous nous sommes contentés, dans cette fonction, de lister les valeurs de ces différents arguments (fixes ou variables), à l'exception du dernier.

*Arguments en nombre variable délimités par une sentinelle*

```
#include <stdio.h>
#include <stdarg.h>

void essai (int par1, char  par2, ...)
{
   va_list adpar ;
   int parv ;
   printf ("premier paramètre : %d\n", par1) ;
   printf ("second paramètre  : %c\n", par2) ;
   va_start (adpar, par2) ;
   while ( (parv = va_arg (adpar, int) ) != -1)
       printf ("argument variable : %d\n", parv) ;
}
```

*Arguments en nombre variable délimités par une sentinelle (suite)*

```
main()
{
   printf ("premier essai\n") ;
   essai (125, 'a', 15, 30, 40, -1) ;
   printf ("\ndeuxième essai\n") ;
   essai (6264, 'S', -1) ;
}
```

```
premier essai
premier paramètre : 125
second paramètre  : a
argument variable : 15
argument variable : 30
argument variable : 40

deuxième essai
premier paramètre : 6264
second paramètre  : S
```

Vous constatez la présence, dans l'en-tête, des deux noms des paramètres fixes `par1` et `par2`, déclarés de manière classique ; les trois points servent à spécifier au compilateur l'existence de paramètres en nombre variable.

La déclaration :

```
va_list adpar
```

précise que `adpar` est un identificateur de liste variable. C'est lui qui nous servira à récupérer, les uns après les autres, les différents arguments variables.

Comme à l'accoutumée, une telle déclaration n'attribue aucune valeur à `adpar`. C'est effectivement la fonction `va_start` qui va permettre de l'initialiser à l'adresse du paramètre variable. Notez bien que cette dernière est déterminée par `va_start` à partir de la connaissance du nom du dernier paramètre fixe.

Le rôle de la fonction `va_arg` est double :

- d'une part, elle fournit comme résultat la valeur trouvée à l'adresse courante fournie par `adpar` (son premier argument), suivant le type indiqué par son second argument (ici `int`).
- d'autre part, elle incrémente l'adresse contenue dans `adpar`, de manière que celle-ci pointe alors sur l'argument variable suivant.

Ici, une instruction `while` nous permet de récupérer les différents arguments variables, sachant que le dernier a pour valeur `-1`.

Enfin, la norme ANSI prévoit que la macro `va_end` doit être appelée avant de sortir de la fonction concernée. Si vous manquez à cette règle, vous courrez le risque de voir un prochain appel à la fonction conduire à un mauvais fonctionnement du programme.

**Remarques**

Les arguments variables peuvent être de types différents, à condition toutefois que la fonction soit en mesure de les connaître, d'une façon ou d'une autre.

Les macros `va_start` et `va_end`, ainsi que la description du type `va_list`, figurent dans le fichier en-tête `stdarg.h` (d'où la directive `#include` correspondante). Cette description utilise une méthode de définition de type (instruction `typedef`) qui ne sera exposée que dans le chapitre relatif aux structures. Notez bien qu'ici vous n'avez pas besoin de savoir ce qui se cache derrière `va_list` pour l'utiliser correctement.

## 12.2 Second exemple

La gestion de la fin de la liste des arguments variables est laissée au bon soin de l'utilisateur ; en effet, il n'existe aucune fonction permettant de connaître le nombre effectif de ces arguments.

Cette gestion peut se faire :

- par sentinelle, comme dans notre précédent exemple,
- par transmission, en argument fixe, du nombre d'arguments variables.

Voici un exemple de fonction utilisant cette seconde technique. Nous n'y avons pas prévu d'autres arguments fixes que celui spécifiant le nombre d'arguments variables.

*Arguments variables dont le nombre est fourni en argument fixe*

```
#include <stdio.h>
#include <stdarg.h>
void essai (int nbpar, ...)
{  va_list adpar ;
   int parv, i ;
   printf ("nombre de valeurs : %d\n", nbpar) ;
   va_start (adpar, nbpar) ;
   for (i=1 ; i<=nbpar ; i++)
     {  parv = va_arg (adpar, int) ;
        printf ("argument variable : %d\n", parv) ;
     }
}
```

*Arguments variables dont le nombre est fourni en argument fixe (suite)*

```
main()
{  printf ("premier essai\n") ;
   essai (3, 15, 30, 40) ;
   printf ("\ndeuxième essai\n") ;
   essai (0) ;
}
```

```
premier essai
nombre de valeurs : 3
argument variable : 15
argument variable : 30
argument variable : 40

deuxième essai
nombre de valeurs : 0
```

# Exercices

Tous ces exercices sont corrigés en fin de volume.

**1)** Écrire :

- une fonction, nommée `f1`, se contentant d'afficher `"bonjour"` (elle ne possèdera aucun argument ni valeur de retour),
- une fonction, nommée `f2`, qui affiche `"bonjour"` un nombre de fois égal à la valeur reçue en argument (`int`) et qui ne renvoie aucune valeur,
- une fonction, nommée `f3`, qui fait la même chose que `f2`, mais qui, de plus, renvoie la valeur (`int`) `0`.

Écrire un petit programme appelant successivement chacune de ces trois fonctions, après les avoir convenablement déclarées sous forme d'un prototype.

**2)** Qu'affiche le programme suivant ?

```
int n=5 ;
main()
{
   void fct (int p) ;
   int n=3 ;
   fct(n) ;
}
void fct(int p)
{
   printf("%d %d", n, p) ;
}
```

**3)** Écrire une fonction qui se contente de comptabiliser le nombre de fois où elle a été appelée en affichant seulement un message de temps en temps, à savoir :

- au premier appel : *** appel 1 fois ***
- au dixième appel : *** appel 10 fois ***
- au centième appel : *** appel 100 fois ***
- et ainsi de suite pour le millième, le dix millième appel...
- On supposera que le nombre maximal d'appels ne peut dépasser la capacité d'un `long`.

**4)** Écrire une fonction récursive calculant la valeur de la « fonction d'Ackermann » `A` définie pour `m>0` et `n>0` par :

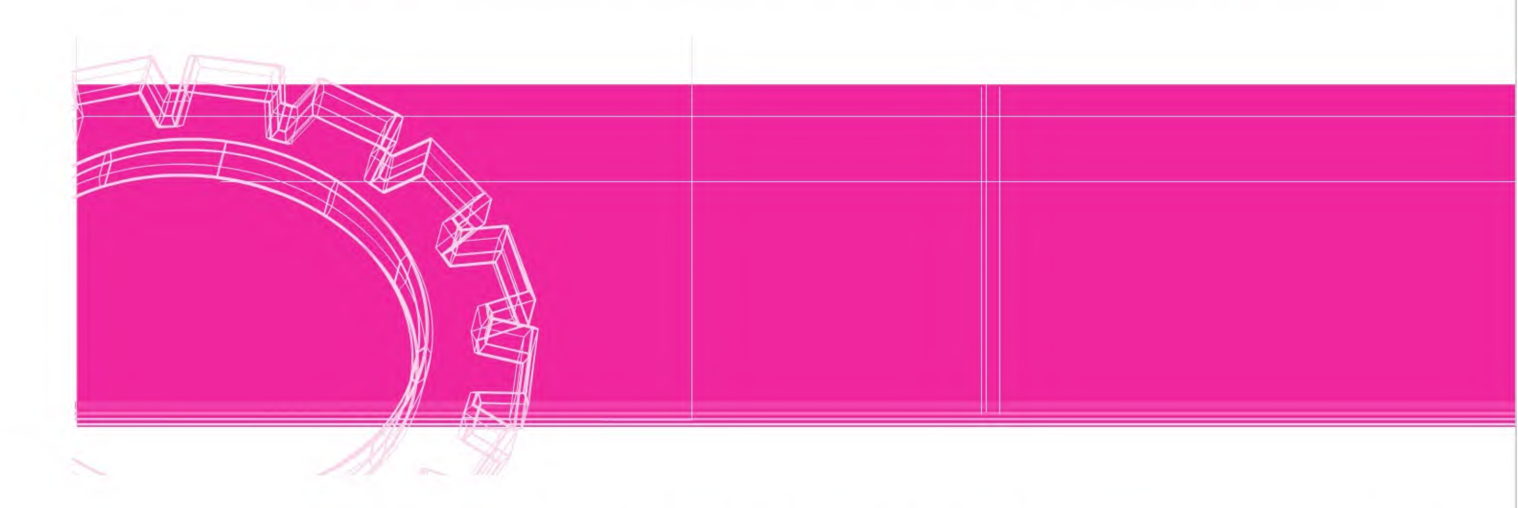
```
A(m,n)  = A(m-1,A(m,n-1))    pour m>0 et n>0
A(0,n)  = n+1    pour n>0
A(m,0)  = A(m-1,1)    pour m>0
```



# Chapitre 7

# Les tableaux et les pointeurs



Comme tous les langages, C permet d'utiliser des **tableaux**. On nomme ainsi un ensemble d'éléments de même type désignés par un identificateur unique ; chaque élément est repéré par un **indice** précisant sa position au sein de l'ensemble.

Par ailleurs, le langage C dispose de **pointeurs**, c'est-à-dire de variables destinées à contenir des adresses d'autres **objets** (variables, fonctions...).

A priori, ces deux notions de tableaux et de pointeurs peuvent paraître fort éloignées l'une de l'autre. Toutefois, il se trouve qu'en C un lien indirect existe entre ces deux notions, à savoir qu'un identificateur de tableau est une « constante pointeur ». Cela peut se répercuter dans le traitement des tableaux, notamment lorsque ceux-ci sont transmis en argument de l'appel d'une fonction.

C'est ce qui justifie que ces deux notions soient regroupées dans un seul chapitre.



## 1 Les tableaux à un indice

### 1.1 Exemple d'utilisation d'un tableau en C

Supposons que nous souhaitions déterminer, à partir de vingt notes d'élèves (fournies en données), combien d'entre elles sont supérieures à la moyenne de la classe.

S'il ne s'agissait que de calculer simplement la moyenne de ces notes, il nous suffirait d'en calculer la somme, en les cumulant dans une variable, au fur et à mesure de leur lecture. Mais, ici, il nous faut à nouveau pouvoir consulter les notes pour déterminer combien d'entre elles sont supérieures à la moyenne ainsi obtenue. Il est donc nécessaire de pouvoir mémoriser ces vingt notes.

Pour ce faire, il paraît peu raisonnable de prévoir vingt variables scalaires différentes (méthode qui, de toute manière, serait difficilement transposable à un nombre important de notes).

Le **tableau** va nous offrir une solution convenable à ce problème, comme le montre le programme suivant.

*Exemple d'utilisation d'un tableau*

```
#include <stdio.h>
main()
{  int i, som, nbm ;
   float moy ;
   int t[20] ;

   for (i=0 ; i<20 ; i++)
      { printf ("donnez la note numéro %d : ", i+1) ;
        scanf ("%d", &t[i]) ;
      }
   for (i=0, som=0 ; i<20 ; i++)  som += t[i] ;
   moy = som / 20 ;
   printf ("\n\n moyenne de la classe : %f\n", moy) ;
   for (i=0, nbm=0 ; i<20 ; i++ )
        if (t[i] > moy) nbm++ ;
   printf ("%d élèves ont plus de cette moyenne", nbm) ;
}
```

La déclaration :

```
int t[20]
```

réserve l'emplacement pour `20` éléments de type `int`. Chaque élément est repéré par sa position dans le tableau, nommée indice. Conventionnellement, en langage C, la première position porte le numéro `0`. Ici, donc, nos indices vont de `0` à `19`. Le premier élément du tableau sera désigné par `t[0]`, le troisième par `t[2]`, le dernier par `t[19]`.

Plus généralement, une notation telle que `t[i]` désigne un élément dont la position dans le tableau est fournie par la valeur de `i`. Elle joue le même rôle qu'une variable scalaire de type `int`.

La notation `&t[i]` désigne l'adresse de cet élément `t[i]` de même que `&n` désignait l'adresse de `n`.



## 1.2 Quelques règles

### a) Les éléments de tableau

Un élément de tableau est une *lvalue*. Il peut donc apparaître à gauche d'un opérateur d'affectation comme dans :

```
t[2] = 5
```

Il peut aussi apparaître comme opérande d'un opérateur d'incrémentation, comme dans :

```
t[3]++              --t[i]
```

En revanche, il n'est pas possible, si `t1` et `t2` sont des tableaux d'entiers, d'écrire `t1 = t2` ; en fait, le langage C n'offre aucune possibilité d'affectations globales de tableaux, comme c'était le cas, par exemple, en Pascal.

### b) Les indices

Un indice peut prendre la forme de n'importe quelle expression arithmétique de type entier (ou caractère, compte tenu des règles de conversion systématique). Par exemple, si `n`, `p`, `k` et `j` sont de type `int`, ces notations sont correctes :

```
t[n-3]
t[3*p-2*k+j%l]
```

Il en va de même, si `c1` et `c2` sont de type `char`, de :

```
t[c1+3]
t[c2-c1]
```

### c) La dimension d'un tableau

La dimension d'un tableau (son nombre d'éléments) ne peut être qu'une **constante** ou une **expression constante**. Ainsi, cette construction :

```
#define N 50
  .....
int t[N] ;
float h[2*N-1] ;
```

est correcte. En revanche, elle ne le serait pas (en C) si `N` était une constante symbolique définie par `const int N=50`, les expressions `N` et `2*N-1` n'étant alors plus calculables par le compilateur (elle sera cependant acceptée en C++).

### d) Débordement d'indice

Aucun contrôle de débordement d'indice n'est mis en place par la plupart des compilateurs. De sorte qu'il est très facile (si l'on peut dire !) de désigner et, donc, de modifier, un emplacement situé avant ou après le tableau.

**Remarque** Pour être efficace, le contrôle d'indice devrait pouvoir se faire, non seulement dans le cas où l'indice est une constante, mais également dans tous les cas où il s'agit d'une expression quelconque. Cela nécessiterait l'incorporation, dans le programme objet, d'instructions supplémentaires assurant cette vérification lors de l'exécution, ce qui conduirait à une perte de temps. Par ailleurs, nous verrons que le problème est rendu encore plus ardu, compte tenu de ce que l'accès à un élément d'un tableau peut également, en C, se faire par le biais d'un pointeur. Pour en comprendre les conséquences, il faut savoir que, lorsque le compilateur rencontre une *lvalue* telle que `t[i]`, il en détermine l'adresse en ajoutant à l'adresse de début du tableau `t`, un décalage proportionnel à la valeur de `i` (et aussi proportionnel à la taille de chaque élément du tableau).

**Remarque C99** La norme C99 autorise que la dimension soit une expression quelconque, pour peu que sa valeur soit calculable au moment où l'on rencontre la déclaration du tableau. Dans ce cas, l'attribution de l'emplacement mémoire correspondant sera réalisé à l'aide d'une technique de « gestion dynamique » semblable à celle présentée au chapitre 11 (mais ici, les instructions nécessaires seront mises en place automatiquement par le compilateur).

# 2 Les tableaux à plusieurs indices

## 2.1 Leur déclaration

Comme tous les langages, C autorise les tableaux à plusieurs indices (on dit aussi à plusieurs dimensions). Par exemple, la déclaration :

```
int t[5][3]
```

réserve un tableau de `15 (5 x 3)` éléments. Un élément quelconque de ce tableau se trouve alors repéré par deux indices comme dans ces notations :

```
t[3][2]     t[i][j]     t[i-3][i+j]
```

Notez bien que, là encore, la notation désignant un élément d'un tel tableau est une *lvalue*. Il n'en ira toutefois pas de même de notations telles que `t[3]` ou `t[j]` bien que, comme nous le verrons un peu plus tard, de telles notations aient un sens en C.

Aucune limitation ne pèse sur le nombre d'indices que peut comporter un tableau. Seules les limitations de taille mémoire liées à un environnement donné risquent de se faire sentir.

## 2.2 Arrangement en mémoire des tableaux à plusieurs indices

Les éléments d'un tableau sont rangés suivant l'ordre obtenu en faisant varier le dernier indice en premier. (Pascal utilise le même ordre, Fortran utilise l'ordre opposé). Ainsi, le tableau `t` déclaré précédemment verrait ses éléments ordonnés comme suit :

```
        t[0][0]
        t[0][1]
```

```
        t[0][2]
        t[1][0]
        t[1][1]
        t[1][2]
          ....
        t[4][0]
        t[4][1]
        t[4][2]
```

Nous verrons que cet ordre a une incidence dans au moins trois circonstances :

- lorsque l'on omet de préciser certaines dimensions d'un tableau,
- lorsque l'on souhaite accéder à l'aide d'un pointeur aux différents éléments d'un tableau,
- lorsque l'un des indices « déborde ». Suivant l'indice concerné et les valeurs qu'il prend, il peut y avoir débordement d'indice sans sortie du tableau. Par exemple, toujours avec notre tableau `t` de `5 x 3` éléments, vous voyez que la notation `t[0][5]` désigne en fait l'élément `t[1][2]`. Par contre, la notation `t[5][0]` désigne un emplacement situé juste au-delà du tableau.

**Remarque**

Bien entendu, les différents points évoqués, dans le paragraphe 1.2, à propos des tableaux à une dimension, restent valables dans le cas des tableaux à plusieurs dimensions.

# 3 Initialisation des tableaux

Comme les variables scalaires, les tableaux peuvent être, suivant leur déclaration, de classe statique ou automatique. Les tableaux de classe statique sont, par défaut, initialisés à zéro ; les tableaux de classe automatique ne sont pas initialisés implicitement.

Il est possible, comme on le fait pour une variable scalaire, d'initialiser (partiellement ou totalement) un tableau lors de sa déclaration. Cette fois, cependant, les valeurs fournies devront obligatoirement être des expressions constantes, et cela quelle que soit la classe d'allocation du tableau concerné (alors que les variables scalaires automatiques pouvaient être initialisées avec des expressions quelconques).

Voici quelques exemples vous montrant comment initialiser un tableau.

## 3.1 Initialisation de tableaux à un indice

La déclaration :

```
int tab[5] = { 10, 20, 5, 0, 3 } ;
```

place les valeurs `10`, `20`, `5`, `0` et `3` dans chacun des cinq éléments du tableau `tab`.

Il est possible de ne mentionner dans les accolades que les premières valeurs, comme dans ces exemples :

```
int tab[5] = { 10, 20 } ;
int tab[5] = { 10, 20, 5 } ;
```

Les valeurs manquantes seront, suivant la classe d'allocation du tableau, initialisées à zéro (statique) ou aléatoires (automatique).

De plus, il est possible d'omettre la dimension du tableau, celle-ci étant alors déterminée par le compilateur par le nombre de valeurs énumérées dans l'initialisation. Ainsi, la première déclaration de ce paragraphe est équivalente à :

```
int tab[] = { 10, 20, 5, 0, 3 } ;
```

**Remarque**

La construction suivante :

```
#define N 10
   ...
int tab[5] = { 2*N-1, N-1, N, N+1, 2*N+1}
```

est correcte. Elle ne le serait pas, en revanche, si l'on définissait `N` comme une constante symbolique entière par `const int N = 10`.

## 3.2 Initialisation de tableaux à plusieurs indices

Voyez ces deux exemples équivalents (nous avons volontairement choisi des valeurs consécutives pour qu'il soit plus facile de comparer les deux formulations) :

```
int tab [3] [4] = { { 1, 2, 3, 4 } ,
                    { 5, 6, 7, 8 },
                    { 9,10,11,12 }   }


int tab [3] [4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 } ;
```

La première forme revient à considérer notre tableau comme formé de trois tableaux de quatre éléments chacun. La seconde, elle, exploite la manière dont les éléments sont effectivement rangés en mémoire et elle se contente d'énumérer les valeurs du tableau suivant cet ordre.

Là encore, à chacun des deux niveaux, les dernières valeurs peuvent être omises. Les déclarations suivantes sont correctes (mais non équivalentes) :

```
int tab [3] [4] = { { 1, 2 } , { 3, 4, 5 } } ;


int tab [3] [4] = { 1, 2 , 3, 4, 5 } ;
```

# 4 Notion de pointeur – Les opérateurs * et &

## 4.1 Introduction

Nous avons déjà été amené à utiliser l'opérateur & pour désigner l'adresse d'une *lvalue*. D'une manière générale, le langage C permet de manipuler des adresses par l'intermédiaire de variables nommées pointeurs.

En guise d'introduction à cette nouvelle notion, considérons les instructions :

```
int * ad ;
int n ;
n = 20 ;
ad = &n ;
*ad = 30 ;
```

La première réserve une variable nommée ad comme étant un pointeur sur des entiers. Nous verrons que * est un opérateur qui désigne le contenu de l'adresse qui le suit. Ainsi, à titre mnémonique, on peut dire que cette déclaration signifie que *ad, c'est-à-dire l'objet d'adresse ad, est de type int ; ce qui signifie bien que ad est l'adresse d'un entier.

L'instruction :

```
ad = &n ;
```

affecte à la variable ad la valeur de l'expression &n. L'opérateur & (que nous avons déjà utilisé avec scanf) est un opérateur unaire qui fournit comme résultat l'adresse de son opérande. Ainsi, cette instruction place dans la variable ad l'adresse de la variable n. Après son exécution, on peut schématiser ainsi la situation :




L'instruction suivante :

```
*ad = 30 ;
```

signifie : affecter à la *lvalue* *ad la valeur 30. Or *ad représente l'entier ayant pour adresse ad (notez bien que nous disons l'entier et pas simplement la valeur car, ne l'oubliez pas, ad est un pointeur sur des entiers). Après exécution de cette instruction, la situation est la suivante :



Bien entendu, ici, nous aurions obtenu le même résultat avec :

```
n = 30 ;
```

## 4.2 Quelques exemples

Voici quelques exemples d'utilisation de ces deux opérateurs. Supposez que nous ayons effectué ces déclarations :

```
    int * ad1, * ad2, * ad ;
    int n = 10, p = 20 ;
```

Les variables `ad1`, `ad2` et `ad` sont donc des pointeurs sur des entiers. Remarquez bien la forme de la déclaration, en particulier, si l'on avait écrit :

```
    int * ad1, ad2, ad ;
```

la variable `ad1` aurait bien été un pointeur sur un entier (puisque `*ad1` est entier) mais `ad2` et `ad` auraient été, quant à eux, des entiers.

Considérons maintenant ces instructions :

```
    ad1 = &n ;
    ad2 = &p ;
    * ad1 = * ad2 + 2 ;
```

Les deux premières placent dans `ad1` et `ad2` les adresses de `n` et `p`. La troisième affecte à `*ad1` la valeur de l'expression :

```
* ad2 + 2
```

Autrement dit, elle place à l'adresse désignée par `ad1` la valeur (entière) d'adresse `ad2`, augmentée de `2`. Cette instruction joue donc ici le même rôle que :

```
n = p + 2 ;
```

De manière comparable, l'expression :

```
* ad1 += 3
```

jouerait le même rôle que :

```
n = n + 3
```

et l'expression :

```
( * ad1 ) ++
```

jouerait le même rôle que `n++` (nous verrons plus loin que, sans les parenthèses, cette expression aurait une signification différente).

**Remarques**

Si `ad` est un pointeur, les expressions `ad` et `*ad` sont des *lvalue* ; autrement dit `ad` et `*ad` sont modifiables. En revanche, il n'en va pas de même de `&ad`. En effet, cette expression désigne, non plus une variable pointeur comme `ad`, mais l'adresse de la variable `ad` telle qu'elle a été définie par le compilateur. Cette adresse est nécessairement fixe et il ne saurait être question de la modifier (la même remarque s'appliquerait à `&n`, où `n` serait une variable scalaire quelconque). D'une manière générale, des expressions telles que :

```
(&ad)++ ou (&p)++
```

seront rejetées à la compilation.

Une déclaration telle que :

```
int * ad
```

réserve un emplacement pour un pointeur sur un entier. Elle ne réserve pas en plus un emplacement pour un tel entier. Cette remarque prendra encore plus d'acuité lorsque les objets pointés seront des chaînes ou des tableaux.

## 4.3 Incrémentation de pointeurs

Jusqu'ici, nous nous sommes contenté de manipuler, non pas les variables pointeurs elles-mêmes, mais les valeurs pointées. Or si une variable pointeur `ad` a été déclarée ainsi :

```
int * ad ;
```

une expression telle que :

```
ad + 1
```

a un sens pour C.

En effet, `ad` est censée contenir l'adresse d'un entier et, pour C, l'expression ci-dessus représente **l'adresse de l'entier suivant**. Certes, dans notre exemple, cela n'a guère d'intérêt car nous ne savons pas avec certitude ce qui se trouve à cet endroit. Mais nous verrons que cela s'avérera fort utile dans le traitement de tableaux ou de chaînes.

Notez bien qu'il ne faut pas confondre un pointeur avec un nombre entier. En effet, l'expression ci-dessus ne représente pas l'adresse de `ad` augmentée de un (octet). Plus précisément, la différence entre `ad+1` et `ad` est ici de `sizeof(int)` octets (n'oubliez pas que l'opérateur `sizeof` fournit la taille, en octets, d'un type donné). Si `ad` avait été déclarée par :

```
double * ad ;
```

cette différence serait de `sizeof(double)` octets.

De manière comparable, l'expression :

```
ad++
```

incrémente l'adresse contenue dans `ad` de manière qu'elle désigne **l'objet** suivant.

Notez bien que des expressions telles que ad+1 ou ad++ sont, en général, valides, quelle que soit l'information se trouvant réellement à l'emplacement correspondant. D'autre part, il est possible d'incrémenter ou de décrémenter un pointeur de n'importe quelle quantité entière. Par exemple, avec la déclaration précédente de ad, nous pourrions écrire ces instructions :

```
ad += 10 ;
ad -= 25 ;
```

**Remarque**

Il existera une exception à ces possibilités, à savoir le cas des pointeurs sur des fonctions, dont nous parlerons plus loin (vous pouvez dès maintenant comprendre qu'incrémenter un pointeur d'une quantité correspondant à la taille d'une fonction n'a pas de sens en soi !).

## 5 Comment simuler une transmission par adresse avec un pointeur

Nous avons vu que le mode de transmission par valeur semblait interdire à une fonction de modifier la valeur de ses arguments effectifs et nous avions mentionné que les pointeurs fourniraient une solution à ce problème.

Nous sommes maintenant en mesure d'écrire une fonction effectuant la permutation des valeurs de deux variables. Voici un programme qui réalise cette opération avec des valeurs entières :

*Utilisation de pointeurs en argument d'une fonction*

```
#include <stdio.h>
main()
{
   void echange (int * ad1, int * ad2) ;
   int a=10, b=20 ;
   printf ("avant appel %d %d\n", a, b) ;
   echange (&a, &b) ;
   printf ("après appel %d %d", a, b) ;
}

void echange (int * ad1, int * ad2)
{  int x ;
   x = * ad1 ;
   * ad1 = * ad2 ;
   * ad2 = x ;
}
```

```
avant appel 10 20
après appel 20 10
```



Les arguments effectifs de l'appel de `echange` sont, cette fois, les adresses des variables `n` et `p` (et non plus leurs valeurs). Notez bien que la transmission se fait toujours par valeur, à savoir que l'on transmet à la fonction `echange` les valeurs des expressions `&n` et `&p`.

Voyez comme, dans `echange`, nous avons indiqué, comme arguments muets, deux variables pointeurs destinées à recevoir ces adresses. D'autre part, remarquez bien qu'il n'aurait pas fallu se contenter d'échanger simplement les valeurs de ces arguments en écrivant (par analogie avec la fonction `echange` du chapitre précédent) :

```
int * x ;
x = ad1 ;
ad1 = ad2 ;
ad2 = x ;
```

Cela n'aurait conduit qu'à échanger (localement) les valeurs de ces deux adresses alors qu'il a fallu échanger les valeurs situées à ces adresses.

**Remarque**

La fonction `echange` n'a aucune raison, ici, de vouloir modifier les valeurs de `ad1` et `ad2`. Nous pourrions préciser dans son en-tête (et, du même coup, dans son prototype) que ce sont en fait des constantes, en l'écrivant ainsi :

```
void echange (int * const ad1, int * const ad2)
```

Notez bien, là encore, la syntaxe de la déclaration des arguments `ad1` et `ad2`. Ainsi, la première s'interprète comme ceci :

`* const ad1` est de type `int`,

`ad1` est donc une constante pointant sur un entier.

Il n'aurait pas fallu écrire :

```
const int * ad1
```

car cela signifierait que :

`int * ad1` est une constante, et que donc :

`ad1` est un pointeur sur un entier constant.

Dans ce dernier cas, la valeur de `ad1` serait modifiable ; en revanche, celle de `*ad1` ne le serait pas et notre programme conduirait à une erreur de compilation.

# 6 Un nom de tableau est un pointeur constant

En langage C, l'identificateur d'un tableau, lorsqu'il est employé seul (sans indices à sa suite), est considéré comme un pointeur (constant) sur le début du tableau. Nous allons en examiner les conséquences en commençant par le cas des tableaux à un indice ; nous verrons en effet que, pour les tableaux à plusieurs indices, il faudra tenir compte du type exact du pointeur en question.

## 6.1 Cas des tableaux à un indice

Supposons, par exemple, que l'on effectue la déclaration suivante :

```
int t[10]
```

La notation `t` est alors totalement équivalente à `&t[0]`.

L'identificateur `t` est considéré comme étant de type pointeur sur le type correspondant aux éléments du tableau, c'est-à-dire, ici, `int *`. Ainsi, voici quelques exemples de notations équivalentes :

```
t+1                &t[1]
t+i                &t[i]
t[i]               * (t+i)
```

Pour illustrer ces nouvelles possibilités de notation, voici plusieurs façons de placer la valeur `1` dans chacun des 10 éléments de notre tableau `t` :

```
     int i ;
     for (i=0 ; i<10 ; i++)
       * (t+i) = 1 ;
```

```
     int i ;
     int * p :
     for (p=t, i=0 ; i<10 ; i++, p++)
       * p = 1 ;
```

Dans la seconde façon, nous avons dû recopier la valeur représentée par `t` dans un pointeur nommé `p`. En effet, il ne faut pas perdre de vue que le symbole `t` représente une adresse constante (`t` est une constante de type pointeur sur des entiers). Autrement dit, une expression telle que `t++` aurait été invalide, au même titre que, par exemple, `3++`. **Un nom de tableau est un pointeur constant ; ce n'est pas une *lvalue*.**

**Remarque**

**Important.** Nous venons de voir que la notation `t[i]` est équivalente à `*(t+i)` lorsque `t` est déclaré comme un tableau. En fait, cela reste vrai, quelle que soit la manière dont `t` a été déclaré. Ainsi, avec :

```
int * t ;
```

les deux notations précédentes resteraient équivalentes. Autrement dit, on peut utiliser `t[i]` dans un programme où `t` est simplement déclaré comme un pointeur (encore faudra-t-il, toutefois, avoir alloué l'espace mémoire nécessaire).

## 6.2 Cas des tableaux à plusieurs indices

Comme pour les tableaux à un indice, l'identificateur d'un tableau, employé seul, représente toujours son adresse de début. Toutefois, si l'on s'intéresse à son type exact, il ne s'agit plus d'un pointeur sur des éléments du tableau. En pratique, ce point n'a d'importance que lorsque l'on effectue des calculs arithmétiques avec ce pointeur (ce qui est assez rare) ou lorsque l'on doit transmettre ce pointeur en argument d'une fonction ; dans ce dernier cas, cependant, nous verrons que le problème est automatiquement résolu par la mise en place de conversions, de sorte qu'on peut ne pas s'en préoccuper.

**À simple titre indicatif**, nous vous présentons ici les règles employées par C, en nous limitant au cas de tableaux à deux indices.

Lorsque le compilateur rencontre une déclaration telle que :

```
int t[3] [4] ;
```

il considère en fait que `t` désigne un tableau de 3 éléments, chacun de ces éléments étant lui-même un tableau de 4 entiers. Autrement dit, si `t` représente bien l'adresse de début de notre tableau `t`, il n'est plus de type `int *` (comme c'était le cas pour un tableau à un indice) mais d'un type « pointeur sur des blocs de 4 entiers », type qui devrait se noter théoriquement (vous n'aurez probablement jamais à utiliser cette notation :

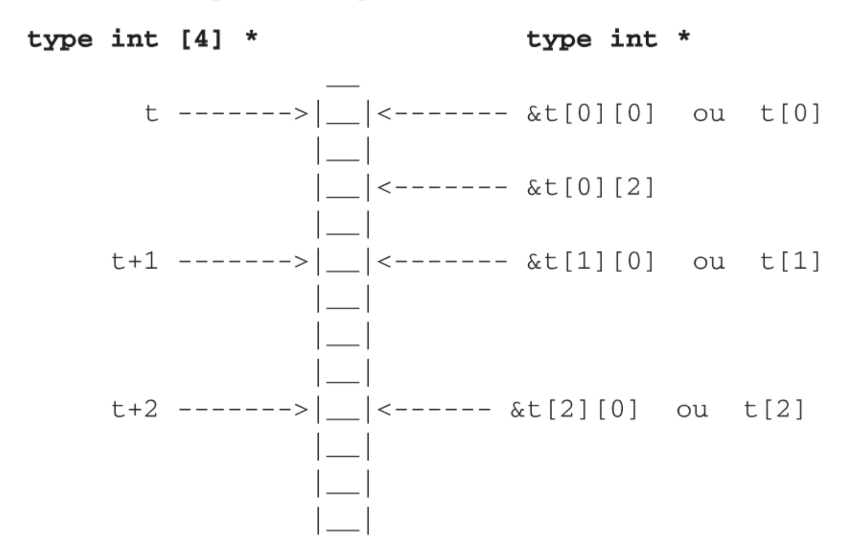
```
int [4] *
```

Dans ces conditions, une expression telle que `t+1` correspond à l'adresse de `t`, augmentée de 4 entiers (et non plus d'un seul !). Ainsi, les notations `t` et `&t[0][0]` correspondent toujours à la même adresse, mais l'incrémentation de `1` n'a pas la même signification pour les deux.

D'autre part, les notations telles que `t[0]`, `t[1]` ou `t[i]` ont un sens. Par exemple, `t[0]` représente l'adresse de début du premier bloc (de 4 entiers) de `t`, `t[1]`, celle du second bloc... Cette fois, il s'agit bien de pointeurs de type `int *`. Autrement, dit les notations suivantes sont totalement équivalentes (elles correspondent à la même adresse et elles sont de même type) :

```
t[0]    &t[0][0]
t[1]    &t[1][0]
```

Voici un schéma récapitulant ce que nous venons de dire.




**Remarque**

*t[1]* est une constante ; ce n'est pas une *lvalue*. L'expression :

```
t[1]++
```

est invalide. Par contre, *t[1][2]* est bien une *lvalue*.

# 7 Les opérateurs réalisables sur des pointeurs

Nous avons déjà vu ce qu'étaient la somme ou la différence d'un pointeur et d'une valeur entière. Nous allons examiner ici les autres opérations réalisables avec des pointeurs.

## 7.1 La comparaison de pointeurs

Il ne faut pas oublier qu'en C un pointeur est défini à la fois par une adresse en mémoire et par un type. On ne pourra donc **comparer que des pointeurs de même type**. Par exemple, voici, en parallèle, deux suites d'instructions réalisant la même action : mise à 1 des 10 éléments du tableau `t` :

```
int t[10] ;                          int t[10] ;
int * p ;                            int i ;
for (p=t ; p<t+10 ; p++)             for (i=0 ; i<10 ; i++)
   *p = 1 ;                              t[i] = 1 ;
```

## 7.2 La soustraction de pointeurs

Là encore, quand deux pointeurs sont **de même type**, leur différence fournit le nombre d'éléments du type en question situés entre les deux adresses correspondantes. L'emploi de cette possibilité est assez rare.

## 7.3 Les affectations de pointeurs et le pointeur nul

Nous avons naturellement déjà rencontré des cas d'affectation de la valeur d'un pointeur à un pointeur de même type. A priori, c'est le seul cas autorisé par le langage C (du moins, tant que l'on ne procède pas à des conversions explicites). Une exception a toutefois lieu en ce qui concerne la valeur entière 0, ainsi que pour le type générique `void *` dont nous parlerons un peu plus loin. Cette tolérance est motivée par le besoin de pouvoir représenter un pointeur nul, c'est-à-dire ne pointant sur rien (c'est le nil du Pascal). Bien entendu, cela n'a d'intérêt que parce qu'il est possible de comparer n'importe quel pointeur (de n'importe quel type) avec ce « pointeur nul ».

D'une manière générale, plutôt que la valeur 0, il est conseillé d'employer la constante `NULL` prédéfinie dans `stdio.h`, et également dans `stddef.h` (bien entendu, elle sera remplacée par la constante entière `0` lors du traitement par le préprocesseur, mais les programmes source en seront néanmoins plus lisibles).

Avec ces déclarations :

```
int * n ;
double * x ;
```

ces instructions seront correctes :

```
n = 0 ;              /* ou mieux */   n = NULL ;
x = 0 ;              /* ou mieux */   x = NULL ;
if (n == 0) ...      /* ou mieux */   if (n == NULL) ...
```

## 7.4 Les conversions de pointeurs

Il n'existe aucune conversion implicite d'un type pointeur dans un autre. En revanche, il est toujours possible de faire appel à l'opérateur de `cast`. D'une manière générale, nous vous conseillons de l'éviter, compte tenu des risques qu'elle comporte. En effet, on pourrait penser qu'une telle conversion revient finalement à ne s'intéresser qu'à l'adresse correspondant à un pointeur, sans s'intéresser au type de l'objet pointé.

Malheureusement, il faut tenir compte de ce que certaines machines imposent aux adresses des objets ce que l'on appelle des « contraintes d'alignement ». Par exemple, un objet de 2 octets sera toujours placé à une adresse paire, tandis qu'un caractère (objet d'un seul octet) pourra être placé (heureusement) à n'importe quelle adresse. Dans ce cas, la conversion d'un `char *` en un `int *` peut conduire soit à l'adresse effective du caractère lorsque celle-ci est paire, soit à une adresse voisine lorsque celle-ci est impaire.

## 7.5 Les pointeurs génériques

En C, un pointeur correspond à la fois à une adresse en mémoire et à un type. Précisément, ce typage des pointeurs peut s'avérer gênant dans certaines circonstances telles que celles où une fonction doit manipuler les adresses d'objets de type non connu (ou, plutôt, susceptible de varier d'un appel à un autre).

Dans certains cas, on pourra satisfaire un tel besoin en utilisant des pointeurs de type `char *`, lesquels, au bout du compte, nous permettront d'accéder à n'importe quel octet de la mémoire. Toutefois, cette façon de procéder implique obligatoirement l'emploi de conversions explicites.

En fait, la norme ANSI a introduit le type pointeur suivant (il n'existait pas dans la définition initiale du langage C, effectuée par Kernighan et Ritchie) :

```
void *
```

Celui-ci désigne un **pointeur sur un objet de type quelconque** (on parle souvent de « pointeur générique »). Il s'agit (exceptionnellement) d'un pointeur sans type.

Une variable de type `void  *` ne peut pas intervenir dans des opérations arithmétiques ; notamment, si `p` et `q` sont de type `void *`, on ne peut pas parler de `p+i` (`i` étant entier) ou de `p-q` ; on ne peut pas davantage utiliser l'expression `p++` ; ceci est justifié par le fait qu'on ne connaît pas la taille des objets pointés. Pour des raisons similaires, il n'est pas possible d'appliquer l'opérateur d'indirection `*` à un pointeur de type `void *`.

Les pointeurs génériques sont théoriquement compatibles avec tous les autres ; autrement dit, les affectations `type * -> void *` sont légales (ce qui ne pose aucun problème) mais les affectations `void * -> type *` (elles seront d'ailleurs illégales en C++) le sont également, ce qui présente les risques évoqués précédemment à propos des contraintes d'alignement.

On notera bien que, lorsqu'il est nécessaire à une fonction de travailler sur les différents octets d'un emplacement de type quelconque, le type `void *` ne convient pas pour décrire les différents octets de cet emplacement et il faudra quand même recourir, à un moment ou à un autre, au type `char *` (mais les conversions `void * --> char *` ne poseront jamais de problème de contraintes d'alignement). Ainsi, pour écrire une fonction qui « met à zéro » un emplacement de la mémoire dont on lui fournit l'adresse et la taille (en octets), on pourrait songer à procéder ainsi :

```
void raz (void * adr, int n)
{
    int i ;
    for (i=0 ; i<n ; i++, adr++)  *adr = 0 ;        // illégal
}
```

Manifestement, ceci est illégal et il faudra utiliser une variable de type `char *` pour décrire notre zone :

```
    void raz (void * adr, int n)
    {
       int i ;
```



```
        char * ad = adr ;
        for (i=0 ; i<n ; i++, ad++)  *ad = 0 ;
    }
```

Voici un exemple d'utilisation de notre fonction `raz` :

```
    void raz (void *, int) ;       /* prototype réduit        */
    int t[10] ;                    /* tableau à mettre à zéro */
    double z ;                     /* double à mettre à zéro  */
      ....
    raz (t, 10*sizeof(int)) ;
    raz (z, sizeof (z)) ;
```

# 8 Les tableaux transmis en argument

Lorsque l'on place le nom d'un tableau en argument effectif de l'appel d'une fonction, on transmet finalement l'adresse du tableau à la fonction, ce qui lui permet d'effectuer toutes les manipulations voulues sur ses éléments, qu'il s'agisse d'utiliser leur valeur ou de la modifier. Voyons quelques exemples pratiques.

## 8.1 Cas des tableaux à un indice

### *a) Premier exemple : tableau de taille fixe*

Voici un exemple de fonction qui met la valeur `1` dans tous les éléments d'un tableau de 10 éléments, l'adresse de ce tableau étant transmise en argument.

*Exemple de tableau transmis en argument d'une fonction (1)*

```
void fct (int t[10])
{
   int i ;
   for (i=0 ; i<10 ; i++) t[i] =1 ;
}
```

Voici deux exemples d'appels possibles de cette fonction :

```
    int t1[10], t2[10] :
      .....
    fct(t1) ;
      ....
    fct(t2) ;
```

L'en-tête de `fct` peut être indifféremment écrit de l'une des manières suivantes :

```
void fct (int t[10])
void fct (int * t)
void fct (int t[])
```

La dernière écriture se justifie par le fait que `t` désigne un argument muet. La réservation de l'emplacement mémoire du tableau dont on recevra ici l'adresse est réalisée par ailleurs dans la fonction appelante (d'ailleurs cette adresse peut changer d'un appel au suivant). D'autre part, la connaissance de la taille exacte du tableau n'est pas indispensable au compilateur ; il est en effet capable de déterminer l'adresse d'un élément quelconque, à partir de son rang et de l'adresse de début du tableau (nous verrons qu'il n'en ira plus de même pour les tableaux à plusieurs indices). Dans ces conditions, on comprend qu'il soit tout à fait possible de ne pas mentionner la dimension du tableau dans l'en-tête de la fonction. En fait, le `10` qui figure dans le premier en-tête n'a d'intérêt que pour le lecteur du programme, afin de lui rappeler la dimension effective du tableau sur lequel travaillait notre fonction.

Par ailleurs, comme d'habitude, quel que soit l'en-tête employé, on peut, dans la définition de la fonction, utiliser indifféremment le formalisme tableau ou le formalisme pointeur. Voici plusieurs écritures possibles de `fct` qui s'accommodent de n'importe lequel des trois en-têtes précédents (elles supposent que `i` a été déclaré de type `int`) :

```
    for (i=0 ; i<10 ; i++)  t[i] = 1 ;

    for (i=0 ; i<10 ; i++, t++) *t = 1 ;

    for (i=0 ; i<10 ; i++)  *(t+i) = 1 ;

    for (i=0 ; i<10 ; i++)  t[i] = 1 ;
```

(ici encore, l'expression `t++` ne pose aucun problème car `t` représente une copie de l'adresse d'un tableau ; `t` est donc bien une *lvalue* et elle peut donc être incrémentée).

Voici enfin une dernière possibilité dans laquelle nous recopions l'adresse `t` dans un pointeur `p` et où nous utilisons les possibilités de comparaison de pointeurs :

```
    int * p ;
    for (p=t ; p<t+10 ; p++) *p = 1 ;
```

### b) Second exemple : tableau de taille variable

Comme nous venons de le voir, lorsqu'un tableau à un seul indice apparaît en argument d'une fonction, le compilateur n'a pas besoin d'en connaître la taille exacte. Il est ainsi facile de réaliser une fonction capable de travailler avec un tableau de dimension quelconque, à condition de lui en transmettre la taille en argument. Voici, par exemple, une fonction qui calcule la somme des éléments d'un tableau d'entiers de taille quelconque :

*Fonction travaillant sur un tableau de taille variable*

```
int som (int t[],int  nb)
{  int s = 0, i ;
   for (i=0 ; i<nb ; i++)
         s += t[i] ;
   return (s) ;
}
```

Voici quelques exemples d'appels de cette fonction :

```
main()
{  int t1[30], t2[15], t3[10] ;
   int s1, s2, s3 ;
      .....
   s1 = som(t1, 30) ;
   s2 = som(t2, 15) + som(t3, 10) ;
    .....
}
```

**Remarque C99** En C99, l'en-tête peut préciser la dimension d'un tableau, à condition que l'argument correspondant apparaisse auparavant. On peut appliquer cette possibilité à la définition de la fonction `som` précédente, à condition d'inverser l'ordre de ses arguments, en procédant ainsi :

```
int som (int nb, int t[nb])
```

L'en-tête suivant serait rejeté :

```
int som (int t[nb], int nb)  /* incorrect, même en C99 */
```

## 8.2 Cas des tableaux à plusieurs indices

### a) Premier exemple : tableau de taille fixe

Voici un exemple d'une fonction qui place la valeur `1` dans chacun des éléments d'un tableau de dimensions `10` et `15` :

*Exemple de transmission en argument d'un tableau à deux dimensions (fixes)*

```
void raun (int t[10][15])
{  int i, j ;
   for (i=0 ; i<10 ; i++)
      for (j=0 ; j<15 ; j++)
         t[i][j] = 1 ;
}
```

Ici, on pourrait, par analogie avec ce que nous avons dit pour un tableau à un indice, utiliser d'autres formes de l'en-tête. Toutefois, il faut bien voir que, pour trouver l'adresse d'un élément quelconque d'un tableau à deux indices, le compilateur ne peut plus se contenter de connaître son adresse de début ; il doit également connaître la seconde dimension du tableau (la première n'étant pas nécessaire compte tenu de la manière dont les éléments sont disposés en mémoire : revoyez le paragraphe 2). Ainsi, l'en-tête de notre fonction aurait pu être `rau (int t[][15])` mais pas `rau (int t[][])`.

En revanche, cette fois, quel que soit l'en-tête utilisé, cette fonction ne convient plus pour un tableau de dimensions différentes de celles pour lesquelles elle a été prévue. Plus précisément, nous pourrons certes toujours l'appeler, comme dans cet exemple :

```
int mat [12][20] ;
     .....
raun (mat) ;
     .....
```

Mais, bien qu'aucun diagnostic ne nous soit fourni par le compilateur, l'exécution de ces instructions placera 150 fois la valeur `1` dans certains des 240 emplacements de `mat`. Qui plus est, avec des tableaux dont la deuxième dimension est inférieure à 15, notre fonction placerait des `1`... en dehors de l'espace attribué au tableau !

**Remarque**

On pourrait songer, par analogie avec ce qui a été fait pour les tableaux à un indice, à mélanger le formalisme pointeur et le formalisme tableau, à la fois dans l'en-tête et dans la définition de la fonction ; cela pose toutefois quelques problèmes que nous allons évoquer dans l'exemple suivant consacré à un tableau de dimensions variables (et dans lequel le formalisme précédent n'est plus applicable).

### b) Second exemple : tableau de dimensions variables

Supposons que nous cherchions à écrire une fonction qui place la valeur 0 dans chacun des éléments de la diagonale d'un tableau carré de taille quelconque. Une façon de résoudre ce problème consiste à adresser les éléments voulus par des pointeurs en effectuant le calcul d'adresse approprié.

*Fonction travaillant sur un tableau carré de taille variable*

```
void diag (int * p, int n)
{
   int i ;
   for (i=0 ; i<n ; i++)
     { * p = 0 ;
       p += n+1 ;
     }
}
```

Notre fonction reçoit donc, en premier argument, l'adresse du premier élément du tableau, sous forme d'un pointeur de type `int *`. Ici, nous avons tenu compte de ce que deux éléments consécutifs de la diagonale sont séparés par `n` éléments. Si, donc, un pointeur désigne un élément de la diagonale, pour pointer sur le suivant il suffit d'incrémenter ce pointeur de `n+1` unités (l'unité étant ici la taille d'un entier).

**Remarques**

Un appel de notre fonction `diag` se présentera ainsi :

```
int t[30] [30] ;
diag (t, 30)
```

Or l'argument effectif `t` est, certes, l'adresse de `t`, mais d'un type pointeur sur des blocs de 10 entiers et non pointeur sur des entiers. En fait, la présence d'un prototype pour `diag` fera qu'il sera converti en un `int *`. Ici, il n'y a aucun risque de modification d'adresse liée à des contraintes d'alignement, car on passe de l'adresse d'un objet de taille `10n` à l'adresse d'un objet de taille `n`. Il n'en irait pas de même avec la conversion inverse.

Cette fonction pourrait également s'écrire en y déclarant un tableau à une seule dimension dont la taille (`n*n`) devrait alors être fournie en argument (en plus de n). Le même mécanisme d'incrémentation de `n+1` s'appliquerait alors, non plus à un pointeur, mais à la valeur d'un indice.

# 9 Utilisation de pointeurs sur des fonctions

En C, comme dans la plupart des autres langages, il n'est pas possible de placer le nom d'une fonction dans une variable. En revanche, on peut y définir une variable destinée à pointer sur une fonction, c'est-à-dire à contenir son adresse.

De plus, en C, le nom d'une fonction (employé seul) est traduit par le compilateur en l'adresse de cette fonction. On retrouve là quelque chose d'analogue à ce qui se passait pour les noms de tableaux, avec toutefois cette différence que les noms de fonctions sont externes (ils subsisteront dans les modules objet).

Ces deux remarques offrent en C des possibilités intéressantes. En voici deux exemples.

## 9.1 Paramétrage d'appel de fonctions

Considérez cette déclaration :

```
int (* adf) (double, int) ;
```

Elle spécifie que :

`(* adf)` est une fonction à deux arguments (de type `double` et `int`) fournissant un résultat de type `int`,

donc que :

> adf est un pointeur sur une fonction à deux arguments (double et int) fournissant un résultat de type int.

Si, par exemple, fct1 et fct2 sont des fonctions ayant les prototypes suivants :

```
int fct1 (double, int) ;
int fct2 (double, int) ;
```

les affectations suivantes ont alors un sens :

```
adf = fct1 ;
adf = fct2 ;
```

Elles placent, dans adf, l'adresse de la fonction correspondante (fct1 ou fct2). Dans ces conditions, il devient possible de programmer un « appel de fonction variable » (c'est-à-dire que la fonction appelée peut varier au fil de l'exécution du programme) par une instruction telle que :

```
(* adf) (5.35, 4) ;
```

Celle-ci, en effet, appelle la fonction dont l'adresse figure actuellement dans adf, en lui transmettant les valeurs indiquées (5.35 et 4). Suivant le cas, cette instruction sera donc équivalente à l'une des deux suivantes :

```
fct1 (5.35, 4) ;
fct2 (5.35, 4) ;
```

## 9.2 Fonctions transmises en argument

Supposez que nous souhaitions écrire une fonction permettant de calculer l'intégrale d'une fonction quelconque suivant une méthode numérique donnée. Une telle fonction que nous supposerons nommée integ posséderait alors un en-tête de ce genre :

```
float integ ( float(*f)(float), ..... )
```

Le premier argument muet correspond ici à l'adresse de la fonction dont on cherche à calculer l'intégrale. Sa déclaration peut s'interpréter ainsi :

> (*f)(float) est de type float,
>
> (*f) est donc une fonction recevant un argument de type float et fournissant un résultat de type float,
>
> f est donc un pointeur sur une fonction recevant un argument de type float et fournissant un résultat de type float.

Au sein de la définition de la fonction integ, il sera possible d'appeler la fonction dont on aura ainsi reçu l'adresse de la façon suivante :

```
(*f) (x)
```

Notez bien qu'il ne faut surtout pas écrire `f(x)`, car `f` désigne ici un pointeur contenant l'adresse d'une fonction, et non pas directement l'adresse d'une fonction.

L'utilisation de la fonction `integ` ne présente pas de difficultés particulières. Elle pourrait se présenter ainsi :

```
main()
{
   float fct1(float), fct2(float) ;
        .....
   res1 = integ (fct1, .....) ;
        .....
   res2 = integ (fct2, .....) ;
        .....
}
```

# Exercices

Tous ces exercices sont corrigés en fin de volume.

**1)** Écrire, de deux façons différentes, un programme qui lit 10 nombres entiers dans un tableau avant d'en rechercher le plus grand et le plus petit :

- en utilisant uniquement le « formalisme tableau »,
- en utilisant le « formalisme pointeur », chaque fois que cela est possible.

**2)** Écrire une fonction qui ne renvoie aucune valeur et qui détermine la valeur maximale et la valeur minimale d'un tableau d'entiers (à un indice) de taille quelconque. Il faudra donc prévoir 4 arguments : le tableau, sa dimension, le maximum et le minimum.

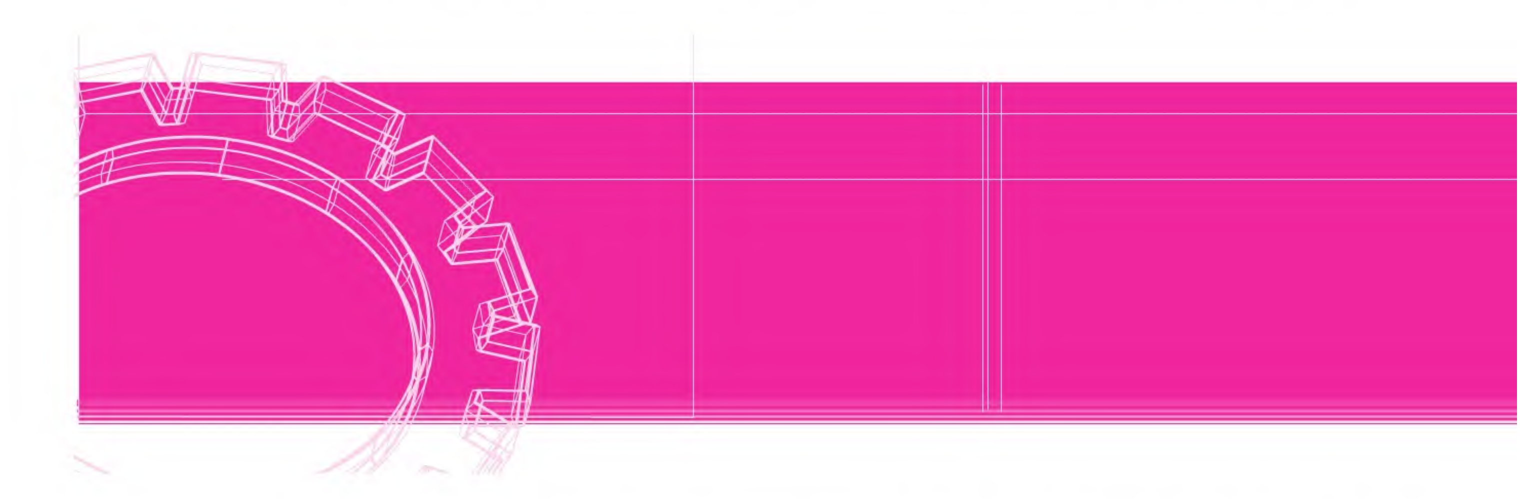
Écrire un petit programme d'essai.

**3)** Écrire une fonction permettant de trier par ordre croissant les valeurs entières d'un tableau de taille quelconque (transmise en argument). Le tri pourra se faire par réarrangement des valeurs au sein du tableau lui-même.

**4)** Écrire une fonction calculant la somme de deux matrices dont les éléments sont de type `double`. Les adresses des trois matrices et leurs dimensions (communes) seront transmises en argument.

# Chapitre 8

# Les chaînes de caractères



Certains langages (Java, Basic, anciennement Turbo Pascal) disposent d'un véritable type chaîne. Les variables d'un tel type sont destinées à recevoir des suites de caractères qui peuvent évoluer, à la fois en contenu et en longueur, au fil du déroulement du programme. Elles peuvent être manipulées d'une manière globale, en ce sens qu'une simple affectation permet de transférer le contenu d'une variable de ce type dans une autre variable de même type.

D'autres langages (Fortran, Pascal standard) ne disposent pas d'un tel type chaîne. Pour traiter de telles informations, il est alors nécessaire de travailler sur des tableaux de caractères dont la taille est nécessairement fixe (ce qui impose à la fois une longueur maximale aux chaînes et ce qui, du même coup, entraîne une perte de place mémoire). La manipulation de telles informations est obligatoirement réalisée caractère par caractère et il faut, de plus, prévoir le moyen de connaître la longueur courante de chaque chaîne.

En langage C, il n'existe pas de véritable type chaîne, dans la mesure où l'on ne peut pas y déclarer des variables d'un tel type. En revanche, il existe une **convention** de représentation des chaînes. Celle-ci est utilisée à la fois :

- par le compilateur pour représenter les chaînes constantes (notées entre doubles quotes) ;
- par un certain nombre de fonctions qui permettent de réaliser :
  - les lectures ou écritures de chaînes ;
  - les traitements classiques tels que concaténation, recopie, comparaison, extraction de sous-chaîne, conversions...

Mais, comme il n'existe pas de variables de type chaîne, il faudra prévoir un emplacement pour accueillir ces informations. Un tableau de caractères pourra faire l'affaire. C'est d'ailleurs ce que nous utiliserons dans ce chapitre. Mais nous verrons plus tard comment créer dynamiquement des emplacements mémoire, lesquels seront alors repérés par des pointeurs.

# 1 Représentation des chaînes

## 1.1 La convention adoptée

En C, une chaîne de caractères est représentée par une suite d'octets correspondant à chacun de ses caractères (plus précisément à chacun de leurs codes), le tout étant terminé par un octet supplémentaire de code nul. Cela signifie que, d'une manière générale, une chaîne de `n` caractères occupe en mémoire un emplacement de `n+1` octets.

## 1.2 Cas des chaînes constantes

C'est cette convention qu'utilise le compilateur pour représenter les « constantes chaîne » (sous-entendu que vous les introduisez dans vos programmes), sous des notations de la forme :

```
"bonjour"
```

De plus, une telle notation sera traduite par le compilateur en un pointeur (sur des éléments de type `char`) sur la zone mémoire correspondante.

Voici un programme illustrant ces deux particularités :

*Convention de représentation des chaînes*

```
#include <stdio.h>
main()
{  char * adr ;                                      bonjour
   adr = "bonjour" ;
   while (*adr)
    { printf ("%c", *adr) ;
      adr++ ;
    }
}
```

La déclaration :

```
char * adr ;
```

réserve simplement l'emplacement pour un pointeur sur un caractère (ou une suite de caractères). En ce qui concerne la constante :

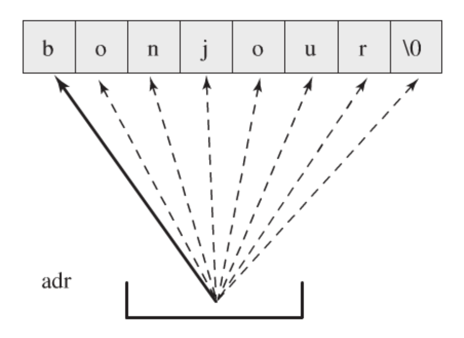
```
"bonjour"
```

le compilateur a créé en mémoire la suite d'octets correspondants mais, dans l'affectation :

```
adr = "bonjour"
```

la notation `bonjour` a comme valeur, non pas la valeur de la chaîne elle-même, mais son adresse ; on retrouve là le même phénomène que pour les tableaux.

Voici un schéma illustrant ce phénomène. La flèche en trait plein correspond à la situation après l'exécution de l'affectation : `adr = "bonjour"` ; les autres flèches correspondent à l'évolution de la valeur de `adr`, au cours de la boucle.




## 1.3 Initialisation de tableaux de caractères

Comme nous l'avons dit, vous serez souvent amené, en C, à placer des chaînes dans des tableaux de caractères.

Mais, si vous déclarez, par exemple :

```
char ch[20] ;
```

vous ne pourrez pas pour autant transférer une chaîne constante dans `ch`, en écrivant une affectation du genre :

```
ch = "bonjour" ;
```

En effet, `ch` est une constante pointeur qui correspond à l'adresse que le compilateur a attribuée au tableau `ch` ; ce n'est pas une *lvalue* ; il n'est donc pas question de lui attribuer une autre valeur (ici, il s'agirait de l'adresse attribuée par le compilateur à la constante chaîne `"bonjour"`).

En revanche, C vous autorise à initialiser votre tableau de caractères à l'aide d'une chaîne constante. Ainsi, vous pourrez écrire :

```
char ch[20] = "bonjour" ;
```

Cela sera parfaitement équivalent à une initialisation de `ch` réalisée par une énumération de caractères (en n'omettant pas le code zéro – noté `\0`) :

```
char ch[20] = { 'b','o','n','j','o','u','r','\0' }
```

N'oubliez pas que, dans ce dernier cas, les 12 caractères non initialisés explicitement seront :

- soit initialisés à zéro (pour un tableau de classe statique) : on voit que, dans ce cas, l'omission du caractère `\0` ne serait (ici) pas grave ;
- soit aléatoires (pour un tableau de classe automatique) : dans ce cas, l'omission du caractère \0 serait nettement plus gênante.

De plus, comme le langage C autorise l'omission de la dimension d'un tableau lors de sa déclaration, lorsqu'elle est accompagnée d'une initialisation, il est possible d'écrire une instruction telle que :

```
char message[] = "bonjour" ;
```

Celle-ci réserve un tableau, nommé `message`, de **8 caractères** (compte tenu du `0` de fin).

## 1.4 Initialisation de tableaux de pointeurs sur des chaînes

Nous avons vu qu'une chaîne constante était traduite par le compilateur en une adresse que l'on pouvait, par exemple, affecter à un pointeur sur une chaîne. Cela peut se généraliser à un tableau de pointeurs, comme dans :

```
char * jour[7] = { "lundi", "mardi", "mercredi", "jeudi",
                   "vendredi", "samedi", "dimanche" } ;
```

Cette déclaration réalise donc à la fois la création des 7 chaînes constantes correspondant aux 7 jours de la semaine et l'initialisation du tableau `jour` avec les 7 adresses de ces 7 chaînes. Voici un exemple employant cette déclaration (nous y avons fait appel, pour l'affichage d'une chaîne, au code de format `%s`, dont nous reparlerons un peu plus loin) :

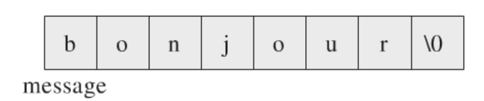
*Initialisation d'un tableau de pointeurs sur des chaînes*

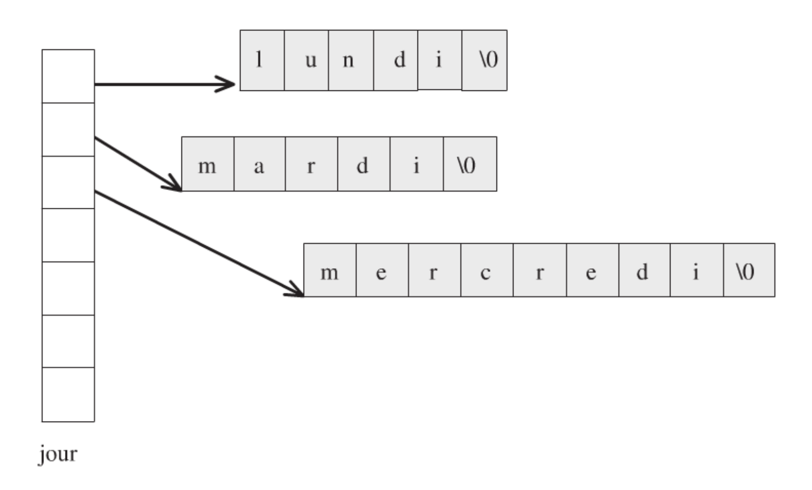
```
main()
{  char * jour[7] = { "lundi", "mardi", "mercredi", "jeudi",
                      "vendredi", "samedi", "dimanche" } ;
   int i ;
   printf ("donnez un entier entre 1 et 7 : ") ;
   scanf ("%d", &i) ;
   printf ("le jour numéro %d de la semaine est %s", i, jour[i-1] ) ;
}
```

```
donnez un entier entre 1 et 7 : 3
le jour numéro 3 de la semaine est mercredi
```

**Remarque**

La situation présentée ne doit pas être confondue avec la précédente. Ici, nous avons affaire à un tableau de sept pointeurs, chacun d'entre eux désignant une chaîne constante (comme le faisait `adr` dans le paragraphe 1.1). Le schéma ci- après récapitule les deux situations.




## 2 Pour lire et écrire des chaînes

Le langage C offre plusieurs possibilités de lecture ou d'écriture de chaînes :

- l'utilisation du code de format %s dans les fonctions `printf` et `scanf` ;
- les fonctions spécifiques de lecture (`gets`) ou d'affichage (`puts`) d'une chaîne (une seule à la fois).

Voyez cet exemple de programme :

*Entrées-sorties classiques de chaînes*

```
#include <stdio.h>
main()
{  char nom[20], prenom[20], ville[25] ;
   printf ("quelle est votre ville : ") ;
   gets (ville) ;
```

*Entrées-sorties classiques de chaînes (suite)*

```
    printf ("donnez votre nom et votre prénom : ") ;
    scanf ("%s %s", nom, prenom) ;
    printf ("bonjour cher %s %s qui habitez à ", prenom, nom) ;
    puts (ville) ;
}
```

```
quelle est votre ville : Paris
donnez votre nom et votre prénom : Dupont Yves
bonjour cher Yves Dupont qui habitez à Paris
```

Les fonctions `printf` et `scanf` permettent de lire ou d'afficher simultanément plusieurs informations de type quelconque. En revanche, `gets` et `puts` ne traitent qu'une chaîne à la fois.

De plus, la délimitation de la chaîne lue ne s'effectue pas de la même façon avec `scanf` et `gets`. Plus précisément :

- avec le code `%s` de `scanf`, on utilise les délimiteurs habituels (l'espace ou la fin de ligne). Cela interdit donc la lecture d'une chaîne contenant des espaces. De plus, le caractère délimiteur n'est pas consommé : il reste disponible pour une prochaine lecture ;
- avec `gets`, seule la fin de ligne sert de délimiteur. De plus, contrairement à ce qui se produit avec `scanf`, ce caractère est effectivement consommé : il ne risque pas d'être pris en compte lors d'une nouvelle lecture.

Dans tous les cas, vous remarquerez que la lecture de n caractères implique le stockage en mémoire de n+1 caractères, car le caractère de fin de chaîne (`\0`) est généré automatiquement par toutes les fonctions de lecture (notez toutefois que le caractère séparateur – fin de ligne ou autre – n'est pas recopié en mémoire).

Ainsi, dans notre précédent programme, il n'est pas possible (du moins pas souhaitable !) que le nom fourni en donnée contienne plus de 19 caractères.

**Remarques**

Dans les appels des fonctions `scanf` et `puts`, les identificateurs de tableau comme `nom`, `prenom` ou `ville` n'ont pas besoin d'être précédés de l'opérateur `&` puisqu'ils représentent déjà des adresses. La norme prévoit toutefois que si l'on applique l'opérateur `&` à un nom de tableau, on obtient l'adresse du tableau. Autrement dit, `&nom` est équivalent à `nom`.

La fonction `gets` fournit en résultat soit un pointeur sur la chaîne lue (c'est donc en fait la valeur de son argument), soit le pointeur nul en cas d'anomalie.

La fonction `puts` réalise un changement de ligne à la fin de l'affichage de la chaîne, ce qui n'est pas le cas de `printf` avec le code de format `%s`.

**Remarques**

Nous nous sommes limité ici aux entrées-sorties conversationnelles. Les autres possibilités seront examinées dans le chapitre consacré aux fichiers.

Si, dans notre précédent programme, l'utilisateur introduit une fin de ligne entre le nom et le prénom, la chaîne affectée à *prenom* n'est rien d'autre que la chaîne vide ! Ceci provient de ce que la fin de ligne servant de délimiteur pour le premier `%s` n'est pas consommée et se trouve donc reprise par le `%s` suivant...

Étant donné que `gets` consomme la fin de ligne servant de délimiteur, alors que le code `%s` de `scanf` ne le fait pas, il n'est guère possible, dans le programme précédent, d'inverser les utilisations de `scanf` et de `gets` (en lisant la `ville` par `scanf` puis le `nom` et le `prénom` par `gets`) : dans ce cas, la fin de ligne non consommée par `scanf` amènerait `gets` à introduire une chaîne vide comme `nom`. D'une manière générale, d'ailleurs, il est préférable, autant que possible, de faire appel à `gets` plutôt qu'au code `%s` pour lire des chaînes.

# 3 Pour fiabiliser la lecture au clavier : le couple `gets` `sscanf`

Nous avons vu, dans le chapitre concernant les entrées-sorties conversationnelles, les problèmes posés par `scanf` en cas de réponse incorrecte de la part de l'utilisateur.

Il est possible de régler la plupart de ces problèmes en travaillant en deux temps :

- lecture d'une chaîne de caractères par `gets` (c'est-à-dire d'une suite de caractères quelconques validés par « return ») ;
- décodage de cette chaîne suivant un format, à l'aide de la fonction **`sscanf`**. En effet, une instruction telle que :

```
sscanf (adresse, format, liste_variables)
```

effectue sur l'emplacement dont on lui fournit l'adresse (premier argument de type `char *`) le même travail que `scanf` effectue sur son tampon. La différence est qu'ici nous sommes maître de ce tampon ; en particulier, nous pouvons décider d'appeler à nouveau `sscanf` sur une nouvelle zone de notre choix (ou sur la même zone dont nous avons modifié le contenu par `gets`), sans être tributaire de la position du pointeur, comme cela était le cas avec `scanf`.

Voici un exemple d'instructions permettant de questionner l'utilisateur jusqu'à ce qu'il ait fourni une réponse satisfaisante

*Contrôle des entrées avec* gets *et* sscanf

```
#include <stdio.h>
#define LG 80
main()
{
   int n, compte ;
   char c ;
   char ligne [LG+1] ;
   do
      { printf ("donnez un entier et un caractère : ") ;
        gets (ligne) ;
        compte = sscanf (ligne, "%d %c", &n, &c) ;
      }
   while (compte < 2 ) ;
   printf ("merci pour %d %c\n", n, c) ;
}
```

```
donnez un entier et un caractère : bof
donnez un entier et un caractère : a 125
donnez un entier et un caractère : 12 bonjour
merci pour 12 b
```

**Remarque**

Nous avons prévu ici des lignes de 80 caractères au maximum. Nous risquons donc de voir le tableau `ligne` « déborder » si l'utilisateur fournit une réponse plus longue. Dans la pratique, on peut augmenter la valeur de `LG`, notamment lorsque, comme c'est souvent le cas, on a affaire à une implémentation où les lignes frappées au clavier ont une taille maximale. Si l'on cherche à réaliser un programme« portable », on préférera une solution qui consiste à remplacer `gets` par `fgets` (`stdin`,...) dont nous parlerons dans le chapitre consacré aux fichiers. La démarche restera identique à celle présentée ici.

# 4 Généralités sur les fonctions portant sur des chaînes

C dispose de nombreuses fonctions de manipulation de chaînes. Avant d'en voir les principales (les autres étant, de toute façon, présentées dans l'annexe), voyons quelques principes généraux.

## 4.1 Ces fonctions travaillent toujours sur des adresses

Tout d'abord, rappelons qu'il n'y a pas de véritable type chaîne en C, mais simplement une convention de représentation. On ne peut donc jamais transmettre la valeur d'une chaîne, mais seulement son adresse, ou plus précisément un pointeur sur son premier caractère. Ainsi, pour comparer deux chaînes, on transmettra à la fonction concernée (ici, `strcmp`) deux pointeurs de type `char *`.

Mieux, pour recopier une chaîne d'un emplacement à un autre, on fournira à la fonction voulue (ici, `strcpy`) l'adresse de la chaîne à copier et l'adresse de l'emplacement où devra se faire la copie. Encore faudra-t-il avoir prévu de disposer de suffisamment de place à cet endroit ! En effet, rien ne permet à la fonction de reconnaître qu'elle a écrit au-delà de ce que vous vouliez. En fait, vous disposerez cependant d'une façon de vous prémunir contre de tels risques ; en effet, toutes les fonctions qui placent ainsi une information (susceptible d'être d'une longueur quelconque) à un emplacement d'adresse donnée possèdent deux variantes : l'une travaillant sans contrôle, l'autre possédant un argument supplémentaire permettant de limiter le nombre de caractères effectivement copiés à l'adresse concernée.

## 4.2 La fonction strlen

La fonction `strlen` fournit en résultat la longueur d'une chaîne dont on lui a transmis l'adresse en argument. Cette longueur correspond tout naturellement au nombre de caractères trouvés depuis l'adresse indiquée jusqu'au premier caractère de code nul, ce caractère n'étant pas pris en compte dans la longueur.

Par exemple, l'expression :

```
strlen ("bonjour")
```

vaudra `7` ; de même, avec :

```
char * adr = "salut" ;
```

l'expression :

```
strlen (adr)
```

vaudra `5`.

### 4.3 Le cas des fonctions de concaténation

Il existe des fonctions dites de concaténation, c'est-à-dire de mise bout à bout de deux chaînes. A priori, de telles fonctions créent une nouvelle chaîne à partir de deux autres. Elles devraient donc recevoir en argument trois adresses ! En fait, C a prévu de se limiter à deux adresses en convenant arbitrairement que la chaîne résultante serait obtenue en ajoutant la seconde à la fin de la première, laquelle se trouve donc détruite en tant que chaîne (en fait, seul son `\0` de fin a disparu...). Là encore, on trouvera deux variantes dont l'une permet de limiter la longueur de la chaîne résultante.

Pour vous familiariser avec cette façon guère naturelle de manipuler les chaînes, nous vous présenterons d'abord en détail les fonctions de concaténation et de copie (ce sont les plus utilisées). Les indications fournies ensuite, ainsi que l'annexe, devraient vous permettre de pouvoir faire appel aux autres sans difficulté.

## 5 Les fonctions de concaténation de chaînes

### 5.1 La fonction `strcat`

Voyez cet exemple :

*Fonction* strcat

```
#include <stdio.h>
#include <string.h>
main()
{
   char ch1[50] = "bonjour" ;
   char * ch2 = " monsieur" ;
   printf ("avant : %s\n", ch1) ;
   strcat (ch1, ch2) ;
   printf ("après : %s", ch1) ;
}
```

```
avant : bonjour
après : bonjour monsieur
```

Notez la différence entre les deux déclarations (avec initialisation) de chacune des deux chaînes `ch1` et `ch2`. La première permet de réserver un emplacement plus grand que la constante chaîne qu'on y place initialement.

L'appel de `strcat` se présente ainsi (nous placerons souvent en regard de la présentation de l'appel d'une fonction le nom du fichier qui en contient le prototype) :

```
strcat ( but, source )          (string.h)
```

Cette fonction recopie la seconde chaîne (`source`) à la suite de la première (`but`), après en avoir effacé le caractère de fin.

**Remarques**

`strcat` fournit en résultat :

- l'adresse de la chaîne correspondant à la concaténation des deux chaînes fournies en argument, lorsque l'opération s'est bien déroulée ; cette adresse n'est rien d'autre que celle de `ch1` (laquelle n'a pas été modifiée – c'est d'ailleurs une constante pointeur),
- le pointeur nul lorsque l'opération s'est mal déroulée.

Il est nécessaire que l'emplacement réservé pour la première chaîne soit suffisant pour y recevoir la partie à lui concaténer.

## 5.2 La fonction strncat

Cette fonction dont l'appel se présente ainsi :

```
strncat (but, source, lgmax)          (string.h)
```

travaille de façon semblable à `strcat` en offrant en outre un contrôle sur le nombre de caractères qui seront concaténés à la chaîne d'arrivée (`but`).

En voici un exemple d'utilisation :

*Fonction* strncat

```
#include <stdio.h>
#include <string.h>
main()
{
   char ch1[50] = "bonjour" ;
   char * ch2 = " monsieur" ;
   printf ("avant : %s\n", ch1) ;
   strncat (ch1, ch2, 6) ;
   printf ("après : %s", ch1) ;
}
```

```
avant : bonjour
après : bonjour monsi
```

Notez bien que le contrôle ne porte pas directement sur la longueur de la chaîne finale. Fréquemment, on déterminera ce nombre maximal de caractères à recopier comme étant la différence entre la taille totale de la zone réceptrice et la longueur courante de la chaîne qui s'y trouve. Cette dernière s'obtiendra par la fonction `strlen` présentée à la section 4.2.

# 6 Les fonctions de comparaison de chaînes

Il est possible de comparer deux chaînes en utilisant l'ordre des caractères définis par leur code.

**a)** La fonction :

```
strcmp ( chaîne1, chaîne2 )
```

compare deux chaînes dont on lui fournit l'adresse et elle fournit une valeur entière définie comme étant :

- positive si `chaîne1 > chaîne2` (c'est-à-dire si `chaîne1` arrive après `chaîne2`, au sens de l'ordre défini par le code des caractères) ;
- nulle si `chaîne1 = chaîne2` (c'est-à-dire si ces deux chaînes contiennent exactement la même suite de caractères) ;
- négative si `chaîne1 < chaîne2`.

Par exemple (quelle que soit l'implémentation) :

```
strcmp ("bonjour", "monsieur")
```

est négatif et :

```
strcmp ("paris2", "paris10")
```

est positif.

**b)** La fonction :

```
strncmp ( chaîne1, chaîne2, lgmax )
```

travaille comme `strcmp` mais elle limite la comparaison au nombre maximal de caractères indiqués par l'entier `lgmax`.

Par exemple :

```
strncmp ("bonjour", "bon", 4)
```

est positif tandis que :

```
strncmp ("bonjour", "bon", 2)
```

vaut zéro.

**c)** Enfin, deux fonctions :

```
stricmp ( chaîne1, chaîne2 )              (string.h)
strnicmp ( chaîne1, chaîne2, lgmax )      (string.h)
```

travaillent respectivement comme `strcmp` et `strncmp`, mais sans tenir compte de la différence entre majuscules et minuscules (pour les seuls caractères alphabétiques).

# 7 Les fonctions de copie de chaînes

**a)** La fonction :

```
strcpy ( but, source )          (string.h)
```

recopie la chaîne située à l'adresse `source` dans l'emplacement d'adresse `destin`. Là encore, il est nécessaire que la taille du second emplacement soit suffisante pour accueillir la chaîne à recopier, sous peine d'écrasement intempestif.

Cette fonction fournit comme résultat l'adresse de la chaîne `but`.

**b)** La fonction :

```
strncpy ( but, source, lgmax )          (string.h)
```

procède de manière analogue à `strcpy`, en limitant la recopie au nombre de caractères précisés par l'expression entière `lgmax`.

Notez bien que, si la longueur de la chaîne source est inférieure à cette longueur maximale, son caractère de fin (`\0`) sera effectivement recopié. Mais, dans le cas contraire, il ne le sera pas. L'exemple suivant illustre les deux situations :

*Fonctions de recopie de chaînes :* strcpy *et* strncpy

```
#include <stdio.h>
#include <string.h>
main()
{
   char ch1[20] = "xxxxxxxxxxxxxxxxxxx" ;
   char ch2[20] ;
   printf ("donnez un mot : ") ;
   gets (ch2) ;
   strncpy (ch1, ch2, 6) ;
   printf ("%s", ch1) ;
}
```

*Fonctions de recopie de chaînes :* strcpy *et* strncpy *(suite)*

```
donnez un mot : bon
bon
________
donnez un mot : bonjour
bonjouxxxxxxxxxxxxx
```

# 8 Les fonctions de recherche dans une chaîne

On trouve, en langage C, des fonctions classiques de recherche de l'occurrence dans une chaîne d'un caractère ou d'une autre chaîne (nommée alors sous-chaîne). Elles fournissent comme résultat **un pointeur de type char *** sur l'information cherchée en cas de succès, et le pointeur nul dans le cas contraire. Voici les principales.

**strchr ( chaîne, caractère )** *(string.h)*

recherche, dans chaîne, la première position où apparaît le caractère mentionné.

**strrchr ( chaîne, caractère )** *(string.h)*

réalise le même traitement que strchr, mais en explorant la chaîne concernée à partir de la fin. Elle fournit donc la dernière occurrence du caractère mentionné.

**strstr ( chaîne, sous-chaîne )** *(string.h)*

recherche, dans chaîne, la première occurrence complète de la sous-chaîne mentionnée.

# 9 Les fonctions de conversion

## 9.1 Conversion d'une chaîne en valeurs numériques

Il existe trois fonctions permettant de convertir une chaîne de caractères en une valeur numérique de type int, long ou double. Ces fonctions ignorent les éventuels espaces de début de chaîne et, à l'image de ce que font les codes de format %d, %ld et %f, utilisent les caractères suivants pour fabriquer une valeur numérique. Le premier caractère invalide arrête l'exploration. En revanche, ici, si aucun caractère n'est exploitable, ces fonctions fournissent un résultat nul.

**atoi ( chaîne )** *(stdlib.h)*

fournit un résultat de type int.



```
atol ( chaîne )            (stdlib.h)
```

fournit un résultat de type `long`.

```
atof ( chaîne )            (stdlib.h)
```

fournit un résultat de type `double`.

Notez que ces fonctions effectuent le même travail que `sscanf` appliquée à une seule variable, avec le code de format approprié. Par exemple (si `n` est de type `int` et `adr` de type `char *`) :

```
n = atoi (adr) ;
```

fait la même chose que :

```
sscanf (adr, "%d", &n) ;
```

### 9.2 Conversion de valeurs numériques en chaîne

La norme ne prévoit pas de fonctions de conversion d'une valeur numérique en chaîne, c'est-à-dire de fonctions jouant le rôle symétrique des fonctions `atoi`, `atol`, `atof` et `atod`. En revanche, elle prévoit une fonction `sprintf`, symétrique de `sscanf`. Elle permet de convertir en chaîne une succession de valeurs numériques, en y incorporant, le cas échéant, d'autres caractères. Par exemple, si `n`, de type `int`, contient `15` et si `p`, de type `float`, contient `785.35` et si `tab` est un tableau de caractères de taille suffisante, l'instruction suivante :

```
sprintf (tab, "%d articles coutent %f8.2 F", n, p) ;
```

placera dans `tab`, la chaîne suivante (elle sera bien terminée par un caractère `\0`) :

```
15 articles coutent   785.35 F
```

## 10 Quelques précautions à prendre avec les chaînes

Dans ce chapitre, nous avons examiné bon nombre des conséquences de la manière artificielle dont le langage C gère les chaînes. Cependant, par souci de clarté, nous nous sommes limité aux situations les plus courantes. Voici ici quelques compléments d'information concernant des situations moins usitées mais dont la méconnaissance peut nuire à la bonne mise au point des programmes.

### 10.1 Une chaîne possède une vraie fin, mais pas de vrai début

Comme nous l'avons vu, il existe effectivement une convention de représentation de la fin d'une chaîne ; en revanche, rien de comparable n'est prévu pour son début. En fait, toute adresse de type `char *` peut toujours faire office d'adresse de début de chaîne.

Par exemple, avec cette déclaration :

```
char * adr = "bonjour" ;
```

une expression telle que :

```
strlen (adr+2)
```

serait acceptée : elle aurait pour valeur `5` (longueur de la chaîne commençant en `adr+2`).

De même, dans l'exemple de programme du paragraphe 5.1, il serait tout à fait possible de remplacer :

```
strcat (ch1, ch2) ;
```

par :

```
strcat (ch1, ch2+4) ;
```

Le programme afficherait alors simplement :

```
bonjoursieur
```

Plus curieusement, si l'on remplace cette fois cette même instruction par :

```
strcat (ch1+2, ch2) ;
```

on obtiendra le même résultat qu'avec le programme initial (`bonjour monsieur`) puisque `ch2` sera toujours concaténée à partir du même `0` de fin !

En revanche, avec :

```
strcat (ch1+10, ch2) ;
```

les choses seraient nettement catastrophiques : on viendrait écraser un emplacement situé en dehors de la chaîne d'adresse `ch1`.

## 10.2 Les risques de modification des chaînes constantes

Nous avons vu que, dans une instruction telle que :

```
char * adr = "bonjour" ;
```

le compilateur remplace la notation `"bonjour"` par l'adresse d'un emplacement dans lequel il a rangé la succession de caractères voulus.

Dans ces conditions, on peut se demander ce qui va se produire si l'on tente de modifier l'un de ces caractères par une banale affectation telle que :

```
*adr = 'x' ;       /* bonjour va-t-il se transformer en xonjour ? */
* (adr+2) = 'x' ; /* bonjour va-t-il se transformer en boxjour ? */
```

A priori, la norme interdit la modification de quelque chose de constant. En pratique, beaucoup de compilateurs l'acceptent, de sorte que l'on aboutit à la modification de notre constante `bonjour` en `xonjour` ou `boxjour` ! Nous pourrions, par exemple, le constater en exécutant une instruction telle que `puts (adr)`.

Signalons qu'une constante chaîne apparaît également dans une instruction telle que :

```
printf ("bonjour") ;
```

Ici, on pourrait penser que sa modification n'est guère possible puisque nous n'avons pas accès à son adresse. Cependant, lorsque cette même constante (`bonjour`) apparaît en plusieurs emplacements d'un programme, certains compilateurs peuvent ne la créer qu'une fois ; dans ces conditions, la chaîne transmise à `printf` peut très bien se trouver modifiée par le processus décrit précédemment...

**Remarque**

Dans une déclaration telle que :

```
char ch[20] = "bonjour" ;
```

il n'apparaît pas de chaîne constante, et ceci malgré la notation employée (`"..."`) laquelle, ici, n'est qu'une facilité d'écriture remplaçant l'initialisation des premiers caractères du tableau `ch`. En particulier, toute modification de l'un des éléments de `ch`, par une instruction telle que :

```
* (ch + 3) = 'x' ;
```

est parfaitement licite (nous n'avons aucune raison de vouloir que le contenu du tableau `ch` reste constant).

# 11 Les arguments transmis à la fonction main

## 11.1 Comment passer des arguments à un programme

La fonction `main` peut récupérer les valeurs des arguments fournis au programme lors de son lancement. Le mécanisme utilisé par l'utilisateur pour fournir ces informations dépend de l'environnement. Il peut s'agir de commandes de menus pour des environnements dits graphiques ou intégrés. Dans les environnements fonctionnant en mode texte (tels DOS ou Unix), il s'agit de valeurs associées à la commande de lancement du programme (d'où le terme d'arguments de la ligne de commande encore utilisé parfois pour décrire ce mécanisme). En voici un exemple où l'on demande l'exécution du programme nommé `test`, en lui transmettant les arguments `arg1`, `arg2` et `arg3` :

```
test arg1 arg2 arg3
```

## 11.2 Comment récupérer ces arguments dans la fonction main

Ces paramètres sont toujours des chaînes de caractères (lorsqu'ils sont fournis dans une commande de lancement du programme, ils sont séparés par des espaces). Leur transmission à la fonction `main` (réalisée par le système) se fait selon les conventions suivantes :

- le premier argument reçu par `main` sera de type `int` et il représentera le nombre total de paramètres fournis dans la ligne de commande (le nom du programme compte lui-même pour un paramètre),
- le second argument reçu par `main` sera l'adresse d'un tableau de pointeurs, chaque pointeur désignant la chaîne correspondant à chacun des paramètres.

Ainsi, en remplaçant l'en-tête de la fonction `main` par celle-ci :

```
main (int nbarg, char * argv[])
```

nous obtiendrons :

- dans `nbarg`, le nombre total de paramètres ;
- à l'adresse `argv[0]`, le premier paramètre, c'est-à-dire le nom du programme (dans notre exemple précédent, il s'agirait donc de la chaîne `test`) ;
- à l'adresse `argv[1]`, le second paramètre (dans notre exemple, il s'agirait donc de la chaîne `arg1`) ;
- etc.

Voici un exemple de programme utilisant ces possibilités. Il est accompagné de trois exemples d'exécution. Nous avons supposé que notre programme se nommait LIGCOM et nous avons noté en gras ce que pourraient être les commandes correspondantes de lancement dans un environnement en mode texte (suivant les implémentations, le nom de programme affiché en résultat pourra différer quelque peu ; par exemple, il pourra être précédé d'une indication de chemin ou de répertoire et suivi d'une extension) :

*Exemple de programme récupérant les arguments de la ligne de commande*

```
#include <stdio.h>
#include <stdarg.h>

main(int nbarg, char * argv[])
{
   int i ;
   printf ("mon nom de programme est : %s\n", argv[0]) ;
   if (nbarg>1) for (i=1 ; i<nbarg ; i++)
                 printf ("argument numéro %d : %s\n", i, argv[i]) ;
       else printf ("pas d'arguments\n") ;
}
```

*Exemple de programme récupérant les arguments de la ligne de commande (suite)*

```
LIGCOM
mon nom de programme est : LIGCOM
pas d'arguments
                    ______________________________________

LIGCOM parametre
mon nom de programme est : LIGCOM
argument numéro 1 : parametre
                    ______________________________________

LIGCOM entree.dat sortie 25CX9
mon nom de programme est : LIGCOM
argument numéro 1 : entree.dat
argument numéro 2 : sortie
argument numéro 3 : 25CX9
```

# Exercices

Tous ces exercices sont corrigés en fin de volume.

**1)** Écrire un programme déterminant le nombre de lettres « e » (minuscules) présentes dans un texte de moins d'une ligne (supposée ne pas dépasser 132 caractères) fourni au clavier.

**2)** Écrire un programme qui supprime toutes les lettres « e » (minuscules) d'un texte de moins d'une ligne (supposée ne pas dépasser 132 caractères) fourni au clavier. Le texte ainsi modifié sera créé, en mémoire, à la place de l'ancien.

**3)** Écrire un programme qui lit au clavier un mot (d'au plus 30 caractères) et qui l'affiche à l'envers.

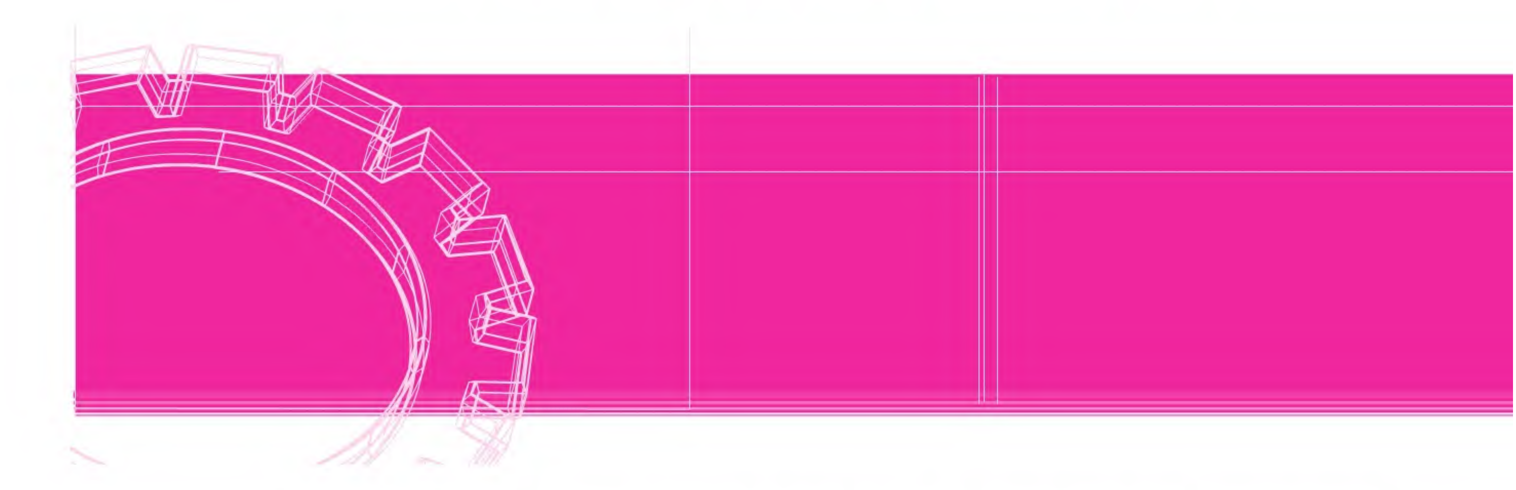
**4)** Écrire un programme qui lit un verbe du premier groupe et qui en affiche la conjugaison au présent de l'indicatif, sous la forme :

```
je chante
tu chantes
il chante
nous chantons
vous chantez
ils chantent
```

Le programme devra vérifier que le mot fourni se termine bien par « er ». On supposera qu'il ne peut comporter plus de 26 lettres et qu'il s'agit d'un verbe régulier. Autrement dit, on admettra que l'utilisateur ne fournira pas un verbe tel que « manger » (le programme afficherait alors : « nous mangons » ).

# Chapitre 9

# Les structures et les énumérations



Nous avons déjà vu comment le tableau permettait de désigner sous un seul nom un ensemble de valeurs de même type, chacune d'entre elles étant repérée par un indice.

La structure, quant à elle, va nous permettre de désigner sous un seul nom un ensemble de valeurs pouvant être de types différents. L'accès à chaque élément de la structure (nommé *champ*) se fera, cette fois, non plus par une indication de position, mais par son nom au sein de la structure.

Quant au type énumération, il s'agit d'un cas particulier de type entier. Sa présentation (tardive) dans ce chapitre ne se justifie que parce que sa déclaration et son utilisation sont très proches de celles du type structure.

# 1 Déclaration d'une structure

Voyez tout d'abord cette déclaration :

```
struct enreg
       {  int numero ;
          int qte ;
          float prix ;
       } ;
```

Celle-ci définit un **modèle de structure** mais ne réserve pas de variables correspondant à cette structure. Ce modèle s'appelle ici `enreg` et il précise le nom et le type de chacun des champs constituant la structure (`numero`, `qte` et `prix`).

Une fois un tel modèle défini, nous pouvons déclarer des variables du type correspondant (souvent, nous parlerons de structure pour désigner une variable dont le type est un modèle de structure).

Par exemple :

```
struct enreg art1 ;
```

réserve un emplacement nommé `art1` « de type `enreg` » destiné à contenir deux entiers et un flottant.

De manière semblable :

```
struct enreg art1, art2 ;
```

réserve deux emplacements `art1` et `art2` du type `enreg`.

**Remarque**

Bien que ce soit peu recommandé, sachez qu'il est possible de regrouper la définition du modèle de structure et la déclaration du type des variables dans une seule instruction comme dans cet exemple :

```
struct enreg
       {  int numero ;
          int qte ;
          float prix ;
       } art1, art2 ;
```

Dans ce dernier cas, il est même possible d'omettre le nom de modèle (`enreg`), à condition, bien sûr, que l'on n'ait pas à déclarer par la suite d'autres variables de ce type.

# 2 Utilisation d'une structure

En C, on peut utiliser une structure de deux manières :

- en travaillant individuellement sur chacun de ses champs ;
- en travaillant de manière globale sur l'ensemble de la structure.

## 2.1 Utilisation des champs d'une structure

Chaque champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant. La désignation d'un champ se note en faisant suivre le nom de la variable structure de l'opérateur « point » (.) suivi du nom de champ tel qu'il a été défini dans le modèle (le nom de modèle lui-même n'intervenant d'ailleurs pas).

Voici quelques exemples utilisant le modèle enreg et les variables art1 et art2 déclarées de ce type.

```
art1.numero = 15 ;
```

affecte la valeur 15 au champ numero de la structure art1.

```
printf ("%e", art1.prix) ;
```

affiche, suivant le code format %e, la valeur du champ prix de la structure art1.

```
scanf ("%e", &art2.prix) ;
```

lit, suivant le code format %e, une valeur qui sera affectée au champ prix de la structure art2. Notez bien la présence de l'opérateur &.

```
art1.numero++
```

incrémente de 1 la valeur du champ numero de la structure art1.

**Remarque**

La priorité de l'opérateur « . » est très élevée, de sorte qu'aucune des expressions ci-dessus ne nécessite de parenthèses (voyez le tableau du chapitre 3, « Les opérateurs et les expressions en langage C »).

## 2.2 Utilisation globale d'une structure

Il est possible d'affecter à une structure le contenu d'une structure définie à partir du **même modèle**. Par exemple, si les structures art1 et art2 ont été déclarées suivant le modèle enreg défini précédemment, nous pourrons écrire :

```
art1 = art2 ;
```

Une telle affectation globale remplace avantageusement :

```
art1.numero = art2.numero ;
art1.qte    = art2.qte ;
art1.prix   = art2.prix ;
```

Notez bien qu'une affectation globale n'est possible que si les structures ont été **définies avec le même nom de modèle** ; en particulier, elle sera impossible avec des variables ayant une structure analogue mais définies sous deux noms différents.

L'opérateur d'affectation et, comme nous le verrons un peu plus loin, l'opérateur d'adresse `&` sont les seuls opérateurs s'appliquant à une structure (de manière globale).

**Remarque**

L'affectation globale n'est pas possible entre tableaux. Elle l'est, par contre, entre structures. Aussi est-il possible, en créant artificiellement une structure contenant un seul champ qui est un tableau, de réaliser une affectation globale entre tableaux.

## 2.3 Initialisations de structures

On retrouve pour les structures les règles d'initialisation qui sont en vigueur pour tous les types de variables, à savoir :

- En l'absence d'initialisation explicite, les structures de classe statique sont, par défaut, initialisées à zéro ; celles possédant la classe automatique ne sont pas initialisées par défaut (elles contiendront donc des valeurs aléatoires).
- Il est possible d'initialiser explicitement une structure lors de sa déclaration. On ne peut toutefois employer que des constantes ou des expressions constantes et cela aussi bien pour les structures statiques que pour les structures automatiques,alors que, pour les variables scalaires automatiques, il était possible d'employer une expression quelconque (on retrouve là les mêmes restrictions que pour les tableaux).

Voici un exemple d'initialisation de notre structure `art1`, au moment de sa déclaration :

```
struct enreg art1 = { 100, 285, 2000 } ;
```

Vous voyez que la description des différents champs se présente sous la forme d'une liste de valeurs séparées par des virgules, chaque valeur étant une constante ayant le type du champ correspondant. Là encore, il est possible d'omettre certaines valeurs.

# 3 Pour simplifier la déclaration de types : définir des synonymes avec typedef

La déclaration `typedef` permet de définir ce que l'on nomme en langage C des *types synonymes*. A priori, elle s'applique à tous les types et pas seulement aux structures. C'est pourquoi nous commencerons par l'introduire sur quelques exemples avant de montrer l'usage que l'on peut en faire avec les structures.

## 3.1 Exemples d'utilisation de typedef

La déclaration :

```
typedef  int  entier ;
```

signifie que `entier` est synonyme de `int`, de sorte que les déclarations suivantes sont équivalentes :

```
    int n, p ;              entier n, p ;
```

De même :

```
typedef  int * ptr ;
```

signifie que `ptr` est synonyme de `int *`. Les déclarations suivantes sont équivalentes :

```
    int * p1, * p2 ;                ptr p1, p2 ;
```

Notez bien que cette déclaration est plus puissante qu'une substitution telle qu'elle pourrait être réalisée par la directive `#define`. Nous n'en ferons pas ici de description exhaustive, et cela d'autant plus que son usage tend à disparaître. À titre indicatif, sachez, par exemple, qu'avec la déclaration :

```
typedef  int vecteur [3] ;
```

les déclarations suivantes sont équivalentes :

```
    int v[3], w[3] ;           vecteur v, w ;
```

## 3.2 Application aux structures

En faisant usage de `typedef`, les déclarations des structures `art1` et `art2` du paragraphe 1 peuvent être réalisées comme suit :

```
     struct enreg
            { int numero ;
              int qte ;
              float prix ;
            } ;
```

```
typedef struct enreg s_enreg ;
s_enreg art1, art2 ;
```

ou encore, plus simplement :

```
typedef struct
        { int numero ;
          int qte ;
          float prix ;
        } s_enreg ;

s_enreg art1, art2 ;
```

Par la suite, nous ne ferons appel qu'occasionnellement à `typedef`, afin de ne pas vous enfermer dans un style de notations que vous ne retrouverez pas nécessairement dans les programmes que vous serez amené à utiliser.

# 4 Imbrication de structures

Dans nos exemples d'introduction des structures, nous nous sommes limité à une structure simple ne comportant que trois champs d'un type de base. Mais chacun des champs d'une structure peut être d'un type absolument quelconque : pointeur, tableau, structure... Il peut même s'agir de pointeurs sur des structures du type de la structure dans laquelle ils apparaissent. Nous en reparlerons dans le chapitre 11, « Gestion dynamique de la mémoire », à propos de la constitution de listes chaînées. De même, un tableau peut être constitué d'éléments qui sont eux-mêmes des structures. Voyons ici quelques situations classiques.

## 4.1 Structure comportant des tableaux

Soit la déclaration suivante :

```
struct personne {  char nom[30] ;
                   char prenom [20] ;
                   float heures [31] ;
                 } employe, courant ;
```

Celle-ci réserve les emplacements pour deux structures nommées `employe` et `courant`. Ces dernières comportent trois champs :

- `nom` qui est un tableau de 30 caractères ;
- `prenom` qui est un tableau de 20 caractères ;
- `heures` qui est un tableau de 31 flottants.

On peut, par exemple, imaginer que ces structures permettent de conserver pour un employé d'une entreprise les informations suivantes :

- nom ;
- prénom ;
- nombre d'heures de travail effectuées pendant chacun des jours du mois courant.

La notation :

```
employe.heures[4]
```

désigne le cinquième élément du tableau `heures` de la structure `employe`. Il s'agit d'un élément de type `float`. Notez que, malgré les priorités identiques des opérateurs `.` et `[]`, leur associativité de gauche à droite évite l'emploi de parenthèses.

De même :

```
employe.nom[0]
```

représente le premier caractère du champ `nom` de la structure `employe`.

Par ailleurs :

```
&courant.heures[4]
```

représente l'adresse du cinquième élément du tableau `heures` de la structure `courant`. Notez que, la priorité de l'opérateur `&` étant inférieure à celle des deux autres, les parenthèses ne sont, là encore, pas nécessaires.

Enfin :

```
courant.nom
```

représente le champ `nom` de la structure `courant`, c'est-à-dire plus précisément l'adresse de ce tableau.

À titre indicatif, voici un exemple d'initialisation d'une structure de type `personne` lors de sa déclaration :

```
struct personne emp = {"Dupont", "Jules", { 8, 7, 8, 6, 8, 0, 0, 8}};
```

## 4.2 Tableaux de structures

Voyez ces déclarations :

```
      struct point { char nom ;
                     int x ;
                     int y ;
                   } ;

      struct point courbe [50] ;
```

La structure `point` pourrait, par exemple, servir à représenter un point d'un plan, point qui serait défini par son nom (caractère) et ses deux coordonnées.

Notez bien que `point` est un nom de modèle de structure, tandis que `courbe` représente effectivement un tableau de 50 éléments du type `point`.

Si `i` est un entier, la notation :

```
courbe[i].nom
```

représente le nom du point de rang `i` du tableau `courbe`. Il s'agit donc d'une valeur de type `char`. Notez bien que la notation :

```
courbe.nom[i]
```

n'aurait pas de sens.

De même, la notation :

```
courbe[i].x
```

désigne la valeur du champ `x` de l'élément de rang `i` du tableau `courbe`.

Par ailleurs :

```
courbe[4]
```

représente la structure de type `point` correspondant au cinquième élément du tableau `courbe`.

Enfin **`courbe`** est un identificateur de tableau, et, comme tel, désigne son adresse de début.

Là encore, voici, à titre indicatif, un exemple d'initialisation (partielle) de notre variable `courbe`, lors de sa déclaration :

```
struct point courbe[50]= { {'A', 10, 25}, {'M', 12, 28},, {'P', 18,2} };
```

## 4.3 Structures comportant d'autres structures

Supposez que, à l'intérieur de nos structures `employe` et `courant` définies dans le paragraphe 4.1, nous ayons besoin d'introduire deux dates : la date d'embauche et la date d'entrée dans le dernier poste occupé. Si ces dates sont elles-mêmes des structures comportant trois champs correspondant au jour, au mois et à l'année, nous pouvons alors procéder aux déclarations suivantes :

```
      struct date
             { int jour ;
               int mois ;
               int annee ;
             } ;
```

```
struct personne
        { char nom[30] ;
          char prenom[20] ;
          float heures [31] ;
          struct date date_embauche ;
          struct date date_poste ;
        } employe, courant ;
```

Vous voyez que la seconde déclaration fait intervenir un modèle de structure (`date`) précédemment défini.

La notation :

```
employe.date_embauche.annee
```

représente l'année d'embauche correspondant à la structure `employe`. Il s'agit d'une valeur de type `int`.

```
courant.date_embauche
```

représente la date d'embauche correspondant à la structure `courant`. Il s'agit cette fois d'une structure de type `date`. Elle pourra éventuellement faire l'objet d'affectations globales comme dans :

```
courant.date_embauche = employe.date_poste ;
```

## 5 À propos de la portée du modèle de structure

À l'image de ce qui se produit pour les identificateurs de variables, la portée d'un modèle de structure dépend de l'emplacement de sa déclaration :

- si elle se situe au sein d'une fonction (y compris, la fonction `main`), elle n'est accessible que depuis cette fonction ;
- si elle se situe en dehors d'une fonction, elle est accessible de toute la partie du fichier source qui suit sa déclaration ; elle peut ainsi être utilisée par plusieurs fonctions.

Voici un exemple d'un modèle de structure nommé `enreg` déclaré à un niveau global et accessible depuis les fonctions `main` et `fct`.

```
struct enreg
        { int numero ;
          int qte ;
          float prix ;
        } ;
```

```
main ()
{   struct enreg x ;
      ....
}
fct ( ....)
{   struct enreg y, z ;
      ....
}
```

En revanche, il n'est pas possible, dans un fichier source donné, de faire référence à un modèle défini dans un autre fichier source. Notez bien qu'il ne faut pas assimiler le nom de modèle d'une structure à un nom de variable ; notamment, il n'est pas possible, dans ce cas, d'utiliser de déclaration `extern`. En effet, la déclaration `extern` s'applique à des identificateurs susceptibles d'être remplacés par des adresses au niveau de l'édition de liens. Or un modèle de structure représente beaucoup plus qu'une simple information d'adresse et il n'a de signification qu'au moment de la compilation du fichier source où il se trouve.

Il est néanmoins toujours possible de placer un certain nombre de déclarations de modèles de structures dans un fichier séparé que l'on incorpore par `#include` à tous les fichiers source où l'on en a besoin. Cette méthode évite la duplication des déclarations identiques avec les risques d'erreurs qui lui sont inhérents.

Le même problème de portée se pose pour les synonymes définis par `typedef`. Les mêmes solutions peuvent y être apportées par l'emploi de `#include`.

## 6 Transmission d'une structure en argument d'une fonction

Jusqu'ici, nous avons vu qu'en C la transmission des argument se fait par valeur, ce qui implique une recopie de l'information transmise à la fonction. Par ailleurs, il est toujours possible de transmettre la valeur d'un pointeur sur une variable, auquel cas la fonction peut, si besoin est, en modifier la valeur. Ces remarques s'appliquent également aux structures (notez qu'il n'en allait pas de même pour un tableau, dans la mesure où la seule chose qu'on puisse transmettre dans ce cas soit la valeur de l'adresse de ce tableau).

### 6.1 Transmission de la valeur d'une structure

Aucun problème particulier ne se pose. Il s'agit simplement d'appliquer ce que nous connaissons déjà. Voici un exemple simple :

*Transmission en argument des valeurs d'une structure*

```
#include <stdio.h>
struct enreg { int a ;
               float b ;
             } ;
main()
{
  struct enreg x ;
  void fct (struct enreg y) ;
  x.a = 1; x.b = 12.5;
  printf ("\navant appel fct : %d %e",x.a,x.b);
  fct (x) ;
  printf ("\nau retour dans main : %d %e", x.a, x.b);
}

void fct (struct enreg s)
{
  s.a = 0; s.b=1;
  printf ("\ndans fct  : %d %e", s.a, s.b);
}
```

```
avant appel fct : 1 1.25000e+01
dans fct   : 0 1.00000e+00
au retour dans main : 1 1.25000e+01
```

Naturellement, les valeurs de la structure `x` sont recopiées localement dans la fonction `fct` lors de son appel ; les modifications de `s` au sein de `fct` n'ont aucune incidence sur les valeurs de `x`.

## 6.2 Transmission de l'adresse d'une structure : l'opérateur ->

Cherchons à modifier notre précédent programme pour que la fonction `fct` reçoive effectivement l'adresse d'une structure et non plus sa valeur. L'appel de `fct` devra donc se présenter sous la forme :

```
fct (&x) ;
```

Cela signifie que son en-tête sera de la forme :

```
void fct (struct enreg * ads) ;
```

Comme vous le constatez, le problème se pose alors d'accéder, au sein de la définition de `fct`, à chacun des champs de la structure d'adresse `ads`. L'opérateur « . » ne convient plus, car il suppose comme premier opérande un nom de structure et non une adresse. Deux solutions s'offrent alors à vous :

- adopter une notation telle que `(*ads).a` ou `(*ads).b` pour désigner les champs de la structure d'adresse `ads` ;
- faire appel à un nouvel opérateur noté `->`, lequel permet d'accéder aux différents champs d'une structure à partir de son adresse de début. Ainsi, au sein de `fct`, la notation `ads -> b` désignera le second champ de la structure reçue en argument ; elle sera équivalente à `(*ads).b`.

Voici ce que pourrait devenir notre précédent exemple en employant l'opérateur noté `->` :

*Transmission en argument de l'adresse d'une structure*

```
#include <stdio.h>
struct enreg { int a ;
               float b ;
             } ;
main()
{
  struct enreg x ;
  void fct (struct enreg *) ;
  x.a = 1; x.b = 12.5;
  printf ("\navant appel fct : %d %e",x.a,x.b);
  fct (&x) ;
  printf ("\nau retour dans main : %d %e", x.a, x.b);
}
void fct (struct enreg * ads)
{
  ads->a = 0 ; ads->b = 1;
  printf ("\ndans fct  : %d %e", ads->a, ads->b);
}
```

```
avant appel fct : 1 1.25000e+01
dans fct  : 0 1.00000e+00
au retour dans main : 0 1.00000e+00
```

# 7 Transmission d'une structure en valeur de retour d'une fonction

Bien que cela soit peu usité, sachez que C vous autorise à réaliser des fonctions qui fournissent en retour la valeur d'une structure. Par exemple, avec le modèle `enreg` précédemment défini, nous pourrions envisager une situation de ce type :

```
struct enreg fct (...)
{   struct enreg s ;       /* structure locale à fct */
     .....
    return s ;             /* dont la fonction renvoie la valeur */
}
```

Notez bien que `s` aura dû soit être créée localement par la fonction (comme c'est le cas ici), soit éventuellement reçue en argument.

Naturellement, rien ne vous interdit, par ailleurs, de réaliser une fonction qui renvoie comme résultat un pointeur sur une structure. Toutefois, il ne faudra pas oublier qu'alors la structure en question ne peut plus être locale à la fonction ; en effet, dans ce cas, elle n'existerait plus dès l'achèvement de la fonction... (mais le pointeur continuerait à pointer sur quelque chose d'inexistant !). Notez que cette remarque vaut pour n'importe quel type autre qu'une structure.

# 8 Les énumérations

Un type énumération est un cas particulier de type entier et donc un type scalaire (ou simple). Son seul lien avec les structures présentées précédemment est qu'il forme, lui aussi, un type défini par le programmeur.

## 8.1 Exemples introductifs

Considérons cette déclaration :

```
enum couleur {jaune, rouge, bleu, vert} ;
```

Elle définit un type énumération nommé `couleur` et précise qu'il comporte quatre valeurs possibles désignées par les identificateurs `jaune`, `rouge`, `bleu` et `vert`. Ces valeurs constituent les constantes du type `couleur`.

Il est possible de déclarer des variables de type `couleur` :

```
enum couleur c1, c2 ;     /* c1 et c2 sont deux variables */
                          /* de type enum couleur         */
```

Les instructions suivantes sont alors tout naturellement correctes :

```
c1 = jaune ;      /* affecte à c1 la valeur jaune            */
c2 = c1 ;         /* affecte à c2 la valeur contenue dans c1 */
```

Comme on peut s'y attendre, les identificateurs correspondant aux constantes du type `couleur` ne sont pas des *lvalue* et ne sont donc pas modifiables :

```
jaune = 3 ;       /* interdit : jaune n'est pas une lvalue */
```

## 8.2 Propriétés du type énumération

### *Nature des constantes figurant dans un type énumération*

Les constantes figurant dans la déclaration d'un type énumération sont des entiers ordinaires. Ainsi, la déclaration précédente :

```
enum couleur {jaune, rouge, bleu, vert} ;
```

associe simplement une valeur de type `int` à chacun des quatre identificateurs cités. Plus précisément, elle attribue la valeur `0` au premier identificateur `jaune`, la valeur `1` à l'identificateur `rouge`, etc. Ces identificateurs sont utilisables en lieu et place de n'importe quelle constante entière :

```
int n ;
long p, q ;
   .....
n = bleu ;                /* même rôle que   n = 2           */
p = vert * q + bleu ;     /* même rôle que   p = 3 * q + 2   */
```

### *Une variable d'un type énumération peut recevoir une valeur quelconque*

Contrairement à ce qu'on pourrait espérer, il est possible d'affecter à une variable de type énuméré n'importe quelle valeur entière (pour peu qu'elle soit représentable dans le type `int`) :

```
enum couleur {jaune, rouge, bleu, vert} ;
enum couleur c1, c2 ;
   .....
c1 = 2 ;     /* même rôle que c1 = bleu ;                 */
c1 = 25 ;    /* accepté, bien que 25 n'appartienne pas au */
             /* type type enum couleur                    */
```

Qui plus est, on peut écrire des choses aussi absurdes que :

```
enum booleen { faux, vrai } ;
enum couleur {jaune, rouge, bleu, vert} ;
enum booleen drapeau ;
enum couleur c ;
   .....
```

```
c = drapeau ;  /* OK bien que drapeau et c ne soit pas d'un même type */
drapeau = 3 * c + 4 ;     /* accepté */
```

### *Les constantes d'un type énumération peuvent être quelconques*

Dans les exemples précédents, les valeurs des constantes attribuées aux identificateurs apparaissant dans un type énumération étaient déterminées automatiquement par le compilateur. Mais il est possible d'influer plus ou moins sur ces valeurs, comme dans :

```
enum couleur_bis { jaune = 5, rouge, bleu, vert = 12, rose } ;
     /* jaune = 5, rouge = 6, bleu = 7, vert = 12, rose = 13 */
```

Les entiers négatifs sont permis comme dans :

```
enum couleur_ter { jaune = -5, rouge, bleu, vert = 12 , rose } ;
     /* jaune = -5, rouge = -4, bleu = -3, vert = 12, rose = 13 */
```

En outre, rien n'interdit qu'une même valeur puisse être attribuée à deux identificateurs différents :

```
enum couleur_ter { jaune = 5, rouge, bleu, vert = 6, noir, violet } ;
   /* jaune = 5, rouge = 6, bleu = 7, vert = 6, noir = 7, violet = 8 */
```

**Remarques**

Comme dans le cas des structures ou des unions, on peut mixer la définition d'un type énuméré et la déclaration de variables utilisant le type. Par exemple, ces deux instructions :

```
enum couleur {jaune, rouge, bleu, vert} ;
enum couleur c1, c2 ;
```

peuvent être remplacées par :

```
enum couleur {jaune, rouge, bleu, vert} c1, c2 ;
```

Dans ce cas, on peut même utiliser un type anonyme, en éliminant l'identificateur de type :

```
enum {jaune, rouge, bleu, vert} c1, c2 ;
```

Cette dernière possibilité présente moins d'inconvénients que dans le cas des structures ou des unions, car aucun problème de compatibilité de type ne risque de se poser.

Compte tenu de la manière dont sont utilisées les structures, il était permis de donner deux noms identiques à des champs de structures différentes. En revanche, une telle possibilité ne peut plus s'appliquer à des identificateurs définis dans une instruction `enum`. Considérez cet exemple :

```
enum couleur {jaune, rouge, bleu, vert} ;
enum bois_carte { rouge, noir } ;  /* erreur : rouge déjà défini */
int rouge ;                        /* erreur : rouge déjà défini */
```

Bien entendu, la portée de tels identificateurs est celle correspondant à leur déclaration (fonction ou partie du fichier source suivant cette déclaration).

# Exercices

Tous ces exercices sont corrigés en fin de volume.

**1)** Écrire un programme qui :

- lit au clavier des informations dans un tableau de structures du type `point` défini comme suit :

```
struct point { int num ;
               float x ;
               float y ;
             }
```

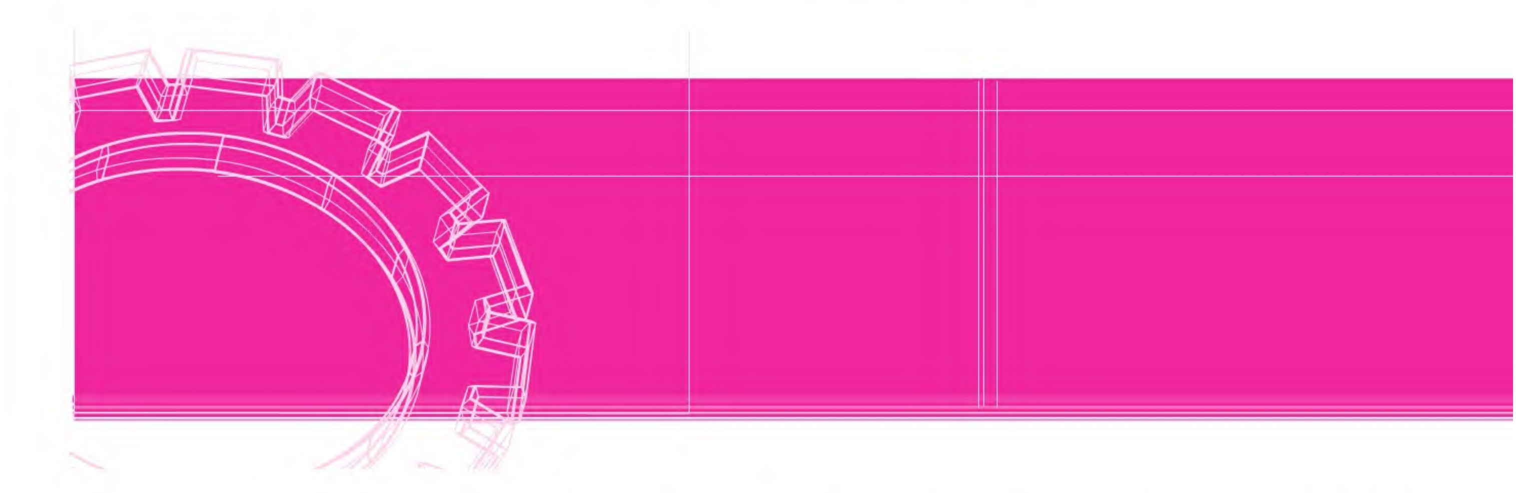
Le nombre d'éléments du tableau sera fixé par une instruction `#define`.

- affiche à l'écran l'ensemble des informations précédentes.

**2)** Réaliser la même chose que dans l'exercice précédent, mais en prévoyant, cette fois, une fonction pour la lecture des informations et une fonction pour l'affichage.

# Chapitre 10

# Les fichiers



Nous avons déjà eu l'occasion d'étudier les « entrées-sorties conversationnelles », c'est-à-dire les fonctions permettant d'échanger des informations entre le programme et l'utilisateur. Nous vous proposons ici d'étudier les fonctions permettant au programme d'échanger des informations avec des fichiers. A priori, ce terme de fichier désigne plutôt un ensemble d'informations situé sur une « mémoire de masse » telle que le disque ou la disquette. Nous verrons toutefois qu'en C, comme d'ailleurs dans d'autres langages, tous les périphériques, qu'ils soient d'archivage (disque, disquette...) ou de communication (clavier, écran, imprimante...), peuvent être considérés comme des fichiers. Ainsi, en définitive, les **entrées-sorties conversationnelles apparaîtront comme un cas particulier de la gestion de fichiers.**

Rappelons que l'on distingue traditionnellement deux techniques de gestion de fichiers :

- l'accès séquentiel consiste à traiter les informations séquentiellement, c'est-à-dire dans l'ordre où elles apparaissent (ou apparaîtront) dans le fichier ;
- l'accès direct consiste à se placer immédiatement sur l'information souhaitée, sans avoir à parcourir celles qui la précèdent.

En fait, pour des fichiers disque (ou disquette), la distinction entre accès séquentiel et accès direct n'a plus véritablement de raison d'être. D'ailleurs, comme vous le verrez, en langage C, vous utiliserez les mêmes fonctions dans les deux cas (exception faite d'une fonction de déplacement de pointeur de fichier). Qui plus est, rien ne vous empêchera de mélanger les deux modes d'accès pour un même fichier. Cependant, pour assurer une certaine progressivité à notre propos, nous avons préféré commencer par vous montrer comment travailler de manière séquentielle.

# 1 Création séquentielle d'un fichier

Voici un programme qui se contente d'enregistrer séquentiellement dans un fichier une suite de nombres entiers saisis au clavier.

*Création séquentielle d'un fichier d'entiers*

```
#include <stdio.h>
main()
{
   char nomfich[21] ;
   int n ;
   FILE * sortie ;

   printf ("nom du fichier à créer : ") ;
   scanf ("%20s", nomfich) ;
   sortie = fopen (nomfich, "w") ;

   do { printf ("donnez un entier : ") ;
        scanf ("%d", &n) ;
        if (n) fwrite (&n, sizeof(int), 1, sortie) ;
      }
  while (n) ;

  fclose (sortie) ;
}
```

Nous avons déclaré un tableau de caractères `nomfich` destiné à contenir, sous forme d'une chaîne, le nom du fichier que l'on souhaite créer.

La déclaration :

```
FILE * sortie ;
```

signifie que `sortie` est un pointeur sur un objet de type `FILE`. Ce nom désigne en fait un modèle de structure défini dans le fichier `stdio.h` (par une instruction `typedef`, ce qui explique l'absence du mot `struct`).

N'oubliez pas que cette déclaration ne réserve qu'un emplacement pour un pointeur. C'est la fonction `fopen` qui créera effectivement une telle structure et qui en fournira l'adresse en résultat.

La fonction `fopen` est ce que l'on nomme une fonction d'ouverture de fichier. Elle possède deux arguments :

- Le nom du fichier concerné, fourni sous forme d'une chaîne de caractères ; ici, nous avons prévu que ce nom ne dépassera pas 20 caractères (le chiffre 21 tenant compte du caractère `\0`) ; notez qu'en général ce nom pourra comporter une information (chemin, répertoire...) permettant de préciser l'endroit où se trouve le fichier.
- Une indication, fournie elle aussi sous forme d'une chaîne, précisant ce que l'on souhaite faire avec ce fichier. Ici, on trouve `w` (abréviation de `write`) qui permet de réaliser une ouverture en écriture. Plus précisément, si le fichier cité n'existe pas, il sera créé par `fopen`. S'il existe déjà, son ancien contenu deviendra inaccessible. Autrement dit, après l'appel de cette fonction, on se retrouve dans tous les cas en présence d'un fichier vide.

Le remplissage du fichier est réalisé par la répétition de l'appel :

```
fwrite (&n, sizeof(int), 1, sortie) ;
```

La fonction `fwrite` possède quatre arguments précisant :

- l'adresse d'un bloc d'informations (ici `&n`) ;
- la taille d'un bloc, en octets : ici `sizeof(int)` ; notez l'emploi de l'opérateur `sizeof` qui assure la portabilité du programme ;
- le nombre de blocs de cette taille que l'on souhaite transférer dans le fichier (ici `1`) ;
- l'adresse de la structure décrivant le fichier (`sortie`).

Notez que, d'une manière générale, `fwrite` permet de transférer plusieurs blocs consécutifs de même taille à partir d'une adresse donnée.

Enfin, la fonction `fclose` réalise ce que l'on nomme une fermeture de fichier. Elle force l'écriture sur disque du tampon associé au fichier. En effet, chaque appel à `fwrite` provoque un entassement d'informations dans le tampon associé au fichier. Ce n'est que lorsque ce dernier est plein qu'il est « vidé » sur disque. Dans ces conditions, on voit qu'après le dernier appel de `fwrite` il est nécessaire de forcer le transfert des dernières informations accumulées dans le tampon.

**Remarques**

On emploie souvent le terme **flux** (en anglais *stream*) pour désigner un pointeur sur une structure de type `FILE`. Ici, par exemple, `sortie` est un flux que la fonction `fopen` aura associé à un certain fichier. D'une manière générale, par souci de simplification, lorsqu'aucune ambiguïté ne sera possible, nous utiliserons souvent le mot fichier à la place de flux.

La fonction `fopen` fournit un pointeur nul en cas d'impossibilité d'ouverture du fichier. Ce sera le cas, par exemple, si l'on cherche à ouvrir en lecture un fichier inexistant ou encore si l'on cherche à créer un fichier sur une disquette saturée.

La fonction `fwrite` fournit le nombre de blocs effectivement écrits. Si cette valeur est inférieure au nombre prévu, cela signifie qu'une erreur est survenue en cours d'écriture. Cela peut être, par exemple, une disquette pleine, mais cela peut se produire également lorsque l'ouverture du fichier s'est mal déroulée (et que l'on n'a pas pris soin d'examiner le code de retour de `fopen`).

## 2 Liste séquentielle d'un fichier

Voici maintenant un programme qui permet de lister le contenu d'un fichier quelconque tel qu'il a pu être créé par le programme précédent.

*Liste séquentielle d'un fichier*

```
#include <stdio.h>
main()
{
   char nomfich[21] ;
   int n ;
   FILE * entree ;

   printf ("nom du fichier à lister : ") ;
   scanf ("%20s", nomfich) ;
   entree = fopen (nomfich, "r") ;

   while ( fread (&n, sizeof(int), 1, entree), ! feof(entree) )
              printf ("\n%d", n) ;

  fclose (entree) ;
}
```

Les déclarations sont identiques à celles du programme précédent. En revanche, on trouve cette fois, dans l'ouverture du fichier, l'indication **r** (abréviation de `read`). Elle précise que le fichier en question ne sera utilisé qu'en lecture. Il est donc nécessaire qu'il existe déjà (nous verrons un peu plus loin comment traiter convenablement le cas où il n'existe pas).

La lecture dans le fichier se fait par un appel de la fonction `fread` :

```
fread (&n, sizeof(int), 1, entree)
```

dont les arguments sont comparables à ceux de `fwrite`. Mais, cette fois, la condition d'arrêt de la boucle est :

```
feof (entree)
```

Celle-ci prend la valeur vrai (c'est-à-dire `1`) lorsque la fin du fichier a été rencontrée. Notez bien qu'il n'est pas suffisant d'avoir lu le dernier octet du fichier pour que cette condition prenne la valeur vrai. Il est nécessaire d'avoir **tenté de lire au-delà** (contrairement à ce qui se

passe, par exemple, en Turbo Pascal, dans lequel la détection de fin de fichier fonctionne en quelque sorte par anticipation) ; c'est ce qui explique que nous ayons examiné cette condition après l'appel de `fread` et non avant.

**Remarques**

On pourrait remplacer la boucle `while` par la construction (moins concise) suivante :

```
do
   { fread (&n, sizeof(int), 1, entree) ;
     if ( !feof(entree) ) printf ("\n%d", n) ;
   }
while ( !feof(entree) ) ;
```

N'oubliez pas que le premier argument des fonctions `fwrite` et `fread` est une adresse. Ainsi, lorsque vous aurez affaire à un tableau, il faudra utiliser simplement son nom (sans le faire précéder de `&`), tandis qu'avec une structure il faudra effectivement utiliser l'opérateur `&` pour en obtenir l'adresse. Dans ce dernier cas, même si l'on ne cherche pas à rendre son programme portable, il sera préférable d'utiliser l'opérateur `sizeof` pour déterminer avec certitude la taille des blocs correspondants.

`fread` fournit le nombre de blocs effectivement lus (et non pas le nombre d'octets lus). Ce résultat peut être inférieur au nombre de blocs demandés soit lorsque l'on a rencontré une fin de fichier, soit lorsqu'une erreur de lecture est apparue. Dans notre précédent exemple d'exécution, `fread` fournit toujours `1`, sauf la dernière fois où elle fournit `0`.

## 3 L'accès direct

Les fonctions `fread` et `fwrite` lisent ou écrivent un certain nombre d'octets dans un fichier, à partir d'une position courante. Cette dernière n'est rien d'autre qu'un « pointeur » dans le fichier, c'est-à-dire un nombre précisant le rang du prochain octet à lire ou à écrire. (Le terme de « pointeur » n'a pas exactement le même sens que celui de pointeur tel qu'il apparaît en langage C. En effet, il ne désigne pas, à proprement parler, une adresse en mémoire, mais un emplacement dans un fichier. Pour éviter des confusions, nous parlerons de « pointeur de fichier »). Après chaque opération de lecture ou d'écriture, ce pointeur se trouve incrémenté du nombre d'octets transférés. C'est ainsi que l'on réalise un accès séquentiel au fichier.

Mais il est également possible d'agir directement sur ce pointeur de fichier à l'aide de la fonction `fseek`. Cela permet ainsi de réaliser des lectures ou des écritures en n'importe quel point du fichier, sans avoir besoin de parcourir toutes les informations qui précèdent. On peut ainsi réaliser ce que l'on nomme généralement un « accès direct ».

## 3.1 Accès direct en lecture sur un fichier existant

*Accès direct en lecture sur un fichier existant*

```
#include <stdio.h>
main()
{  char nomfich[21] ;
   int n ;
   long num ;
   FILE * entree ;
   printf ("nom du fichier à consulter : ") ;
   scanf ("%20s", nomfich) ;
   entree = fopen (nomfich, "r") ;
   while ( printf (" numéro de l'entier recherché : "),
           scanf ("%ld", &num), num )
      { fseek (entree, sizeof(int)*(num-1), SEEK_SET) ;
        fread (&n, sizeof(int), 1, entree) ;
        printf ("  valeur : %d \n", n) ;
      }
   fclose (entree) ;
```

Le programme ci-dessus permet d'accéder à n'importe quel entier d'un fichier du type de ceux que pouvait créer notre programme de la section 2.1.

La principale nouveauté réside essentiellement dans l'appel de la fonction `fseek` :

```
fseek ( entree, sizeof(int)*(num-1), SEEK_SET) ;
```

Cette dernière possède trois arguments :

- le fichier concerné (désigné par le pointeur sur une structure de type `FILE`, tel qu'il a été fourni par `fopen`) ;
- un entier de type `long` spécifiant la valeur que l'on souhaite donner au pointeur de fichier. Il faut noter que l'on dispose de trois manières d'agir effectivement sur le pointeur, le choix entre les trois étant fait par l'argument suivant ;
- le choix du mode d'action sur le pointeur de fichier : il est défini par une constante entière. Les valeurs suivantes sont prédéfinies dans `<stdio.h>` :
  - `SEEK_SET` (en général `0`) : le second argument désigne un déplacement (en octets) depuis le **début du fichier** ;
  - `SEEK_CUR` (en général `1`) : le second argument désigne un déplacement exprimé à partir de la **position courante** ; il s'agit donc en quelque sorte d'un déplacement relatif dont la valeur peut, le cas échéant, être négative ;
  - `SEEK_END` (en général `2`) : le second argument désigne un déplacement depuis la **fin du fichier**.

**Remarque**

En général, ces constantes auront la valeur indiquée (0, 1 et 2). Toutefois, la norme n'impose pas précisément ces valeurs ; elle se contente d'imposer l'existence des trois constantes symboliques que l'on aura donc intérêt à utiliser si l'on souhaite assurer la portabilité des programmes concernés.

Ici, il s'agit de donner au pointeur de fichier une valeur correspondant à l'emplacement d'un entier (`sizeof(int)` octets) dont l'utilisateur fournit le rang. Il est donc naturel de donner au troisième argument la valeur `0`. Notez, au passage, la formule :

```
sizeof(int) * (num-1)
```

qui se justifie par le fait que nous avons convenu que, pour l'utilisateur, le premier entier du fichier porterait le rang `1` et non `0`.

## 3.2 Les possibilités de l'accès direct

Outre les possibilités de consultation immédiate qu'il procure, l'accès direct facilite et accélère les opérations de mise à jour d'un fichier.

Mais, de surcroît, l'accès direct permet de remplir un fichier de façon quelconque. Ainsi, nous pourrions constituer notre fichier d'entiers en laissant l'utilisateur fournir ces entiers dans l'ordre de son choix, comme dans cet exemple de programme :

*Création d'un fichier en accès direct*

```
#include <stdio.h>
main()
{  char nomfich[21] ;
   FILE * sortie ;
   long num ;
   int n ;

   printf ("nom fichier : ") ;
   scanf  ("%20s",nomfich) ;
   sortie = fopen (nomfich, "w") ;

   while (printf("\nrang de l'entier : "), scanf("%ld",&num), num)
         {  printf ("valeur de l'entier : ") ;
            scanf ("%d", &n) ;
            fseek (sortie, sizeof(int)*(num-1), SEEK_SET)   ;
            fwrite (&n, sizeof(int), 1, sortie) ;
         }

   fclose(sortie) ;
```

Or il faut savoir qu'avec beaucoup de systèmes, dès que vous écrivez le énième octet d'un fichier, il y a automatiquement réservation de la place de tous les octets précédents ; leur contenu, par contre, doit être considéré comme étant aléatoire. De toute façon, tous les systèmes réservent toujours la place d'un nombre minimal d'octets, de sorte que le problème évoqué existe toujours, au moins pour certains octets du fichier.

Dans ces conditions, vous voyez que, à partir du moment où rien n'impose à l'utilisateur de ne pas « laisser de trous » lors de la création du fichier, il faudra être en mesure de repérer ces trous lors d'éventuelles consultations ultérieures du fichier. Plusieurs techniques existent à cet effet :

- on peut, par exemple, avant d'exécuter le programme précédent, commencer par initialiser tous les emplacements du fichier à une *valeur spéciale*, dont on sait qu'elle ne pourra pas apparaître comme valeur effective ;
- on peut aussi gérer une table des emplacements inexistants, cette table devant alors être conservée (de préférence) dans le fichier lui-même.

D'autre part, il faut bien voir que l'accès direct n'a d'intérêt que lorsque l'on est en mesure de fournir le rang de l'emplacement concerné. Ce n'est pas toujours possible. Ainsi, si l'on considère ne serait-ce qu'un simple fichier de type répertoire téléphonique, on voit qu'en général on cherchera à accéder à une personne par son nom plutôt que par son numéro d'ordre dans le fichier. Cette contrainte qui semble imposer une recherche séquentielle peut être contournée par la création de ce que l'on nomme un *index*, c'est-à-dire une table de correspondance entre un nom d'individu et sa position dans le fichier.

Nous n'en dirons pas plus sur ces méthodes spécifiques de gestion de fichiers qui sortent du cadre de cet ouvrage.

## 3.3 En cas d'erreur

### *a) Erreur de pointage*

Il faut bien voir que le positionnement dans le fichier se fait sur un octet de rang donné, et non, comme on pourrait le préférer, sur un bloc (ou enregistrement) de rang donné. D'ailleurs, n'oubliez pas qu'en général cette notion d'enregistrement n'est pas exprimée de manière intrinsèque au sein du fichier. Ainsi, dans notre programme précédent, vous pourriez, par mégarde, utiliser la formule suivante :

```
sizeof(int) * num -1
```

laquelle vous positionnerait systématiquement « à cheval » entre le dernier octet d'un entier et le premier du suivant. Bien entendu, les résultats obtenus seraient quelque peu fantaisistes.

Cette remarque prend encore plus d'acuité lorsque vous créez un fichier à partir de structures. Dans ce cas, nous ne saurions trop vous conseiller d'avoir systématiquement recours à l'opérateur `sizeof` pour déterminer la taille réelle de ces structures.

### b) Tentative de positionnement hors fichier

Lorsque l'on accède ainsi directement à l'information d'un fichier, le risque existe de tenter de se positionner... en dehors du fichier. En principe, la fonction `fseek` fournit :

- la valeur `0` lorsque le positionnement s'est déroulé correctement ;
- une valeur quelconque dans le cas contraire.

Toutefois, beaucoup d'implémentations ne respectent pas la norme à ce sujet. Dans ces conditions, il nous paraît plus raisonnable de programmer une protection efficace en déterminant, en début de programme, la taille effective du fichier à consulter. Pour cela, il suffit de vous positionner en fin de fichier avec `fseek`, puis de faire appel à la fonction `ftell` qui restitue la position courante du pointeur de fichier. Ainsi, dans notre précédent programme, nous pourrions introduire les instructions :

```
long taille ;
    .....
fseek (entree, 0, SEEK_END) ;
taille = ftell (entree) ;
```

Il suffit alors de vérifier que la position de l'enregistrement demandé (ici : `sizeof(int*(num-1))`) est bien inférieure à la valeur de `taille` pour éviter tout problème.

## 4 Les entrées-sorties formatées et les fichiers de texte

Nous venons de voir que les fonctions `fread` et `fwrite` réalisent un transfert d'information (entre mémoire et fichier) que l'on pourrait qualifier de brut, dans le sens où il se fait sans aucune transformation de l'information. Les octets qui figurent dans le fichier sont des copies conformes de ceux qui apparaissent en mémoire.

Mais, en langage C, il est également possible d'accompagner ces transferts d'information d'opérations de formatage analogues à celles que réalisent `printf` ou `scanf`.

Les fichiers concernés par ces opérations de formatage sont alors ce que l'on a coutume d'appeler des « fichiers de type texte » ou encore des « fichiers de texte ». Ce sont des fichiers que vous pouvez manipuler avec un éditeur ou un traitement de texte quelconques ou, encore plus simplement, lister par les commandes appropriées du système d'exploitation (`TYPE` ou `PRINT` sous DOS, `pr` ou `more` sous UNIX...).

Dans de tels fichiers, chaque octet représente un caractère. Généralement, on y trouve des caractères de fin de ligne (`\n`), de sorte qu'ils apparaissent comme une suite de lignes. Les fonctions permettant de travailler avec des fichiers de texte ne sont rien d'autre qu'une généralisation de celles que nous avons déjà rencontrées pour les entrées-sorties conversationnelles.

Nous nous contenterons donc d'en fournir une brève liste :

```
fscanf ( fichier, format, liste_d'adresses )
fprintf ( fichier, format, liste_d'expressions )
fgetc ( fichier )
fputc ( entier, fichier )
fgets ( chaîne, lgmax, fichier )
fputs ( chaîne, fichier )
```

La signification de leurs arguments est la même que pour les fonctions conversationnelles correspondantes. Seule `fgets` comporte un argument entier (`lgmax`) de contrôle de longueur. Il précise le nombre maximal de caractères (y compris le `\0` de fin) qui seront placés dans la chaîne.

Leur valeur de retour est la même que pour les fonctions conversationnelles. Cependant, il nous faut apporter quelques indications supplémentaires qui ne se justifiaient pas pour des entrées-sorties conversationnelles (mais qui auraient un intérêt en cas de simple « redirection » des entrées-sorties), à savoir que la valeur de retour fournie par `fgetc` est du type `int` (et non, comme on pourrait le croire, de type `char`). Lorsque la fin de fichier est atteinte, cette fonction fournit la valeur `EOF` (constante prédéfinie dans `<stdio.h>` – en général `-1`). La fin de fichier n'est détectée que lorsque l'on cherche à lire un caractère alors qu'il n'y en a plus de disponible, et non pas, dès que l'on a lu le dernier caractère (ce qui, ici, serait assez mal approprié puisque alors on ne pourrait obtenir à la fois le code de ce caractère et une indication de fin de fichier). D'autre part, notez bien que cette convention fait, en quelque sorte, double emploi avec la fonction `feof`.

D'une manière générale, toutes les fonctions présentées ci-dessus fournissent une valeur de retour bien définie en cas de fin de fichier ou d'erreur. Vous trouverez tous les détails utiles dans l'annexe.

**Remarque**

**Important.** A priori, on peut toujours dire que n'importe quel fichier, quelle que soit la manière dont l'information y a été représentée, peut être considéré comme une suite de caractères. Bien entendu, si l'on cherche à lister, par exemple, le contenu d'un fichier tel que celui créé dans la section 2.1 (suite d'entiers), le résultat risque d'être sans signification (on obtiendra une suite de caractères apparemment quelconques, sans rapport aucun avec les nombres enregistrés).

Mais, sans aller jusqu'à le lister, on peut se demander s'il ne serait pas possible de le recopier, à l'aide d'une répétition de `fgetc` et de `fputc`. Or cela semble effectivement possible puisque ces fonctions se contentent de prélever un caractère (donc un octet) et de le recopier **tel quel**. Ainsi, quel que soit le contenu de l'octet lu, on le retrouvera dans le fichier de sortie.

**En réalité**, cela n'est que partiellement vrai car **certains environnements distinguent les fichiers de texte des autres** (qu'ils appellent parfois « fichiers binaires », alors qu'au bout du compte tout fichier est binaire !) ; plus précisément, lors de l'ouverture du fichier, on peut spécifier si l'on souhaite ou non considérer le contenu du fichier comme du texte. Cette distinction est en fait motivée par le fait que le caractère de fin de ligne (`\n`) possède, dans ces environnements, une représentation particulière obtenue par la succession de deux caractères (retour



chariot `\r`, suivi de fin de ligne `\n`). Or, dans ce cas, pour qu'un programme C puisse ne voir qu'un seul caractère de fin de ligne et qu'il s'agisse bien de `\n`, il faut opérer un traitement particulier consistant à :

- remplacer chaque occurrence de ce couple de caractères par `\n`, dans le cas d'une lecture,
- remplacer chaque demande d'écriture de `\n` par l'écriture de ce couple de caractères.

Bien entendu, de telles substitutions ne doivent pas être réalisées sur de vrais fichiers binaires. Il faut donc bien pouvoir opérer une distinction au sein du programme. Cette distinction se fait au moment de l'ouverture du fichier, en ajoutant l'une des lettres `t` (pour « texte ») ou `b` (pour « binaire ») au mode d'ouverture.

En général, dans les implémentations où l'on distingue les fichiers de texte des autres, les fonctions d'entrées-sorties formatées refusent de travailler avec un fichier qui n'a pas été spécifié de ce type lors de son ouverture.

## 5 Les différentes possibilités d'ouverture d'un fichier

Dans nos précédents exemples, nous n'avons utilisé que les modes `w` et `r`. Nous vous fournissons ici la liste des différentes possibilités offertes par `fopen`.

**`r`** : *lecture* seulement ; le fichier doit exister.

**`w`** : *écriture* seulement. Si le fichier n'existe pas, il est créé. S'il existe, son (ancien) contenu est perdu.

**`a`** : *écriture en fin de fichier* (append). Si le fichier existe déjà, il sera étendu. S'il n'existe pas, il sera créé – on se ramène alors au mode **`w`**.

**`r+`** : *mise à jour* (lecture et écriture). Le fichier doit exister. Notez qu'alors il n'est pas possible de réaliser une lecture à la suite d'une écriture ou une écriture à la suite d'une lecture, sans positionner le pointeur de fichier par `fseek`. Il est toutefois possible d'enchaîner plusieurs lectures ou écritures consécutives (de façon séquentielle).

**`w+`** : *création pour mise à jour*. Si le fichier existe, son (ancien) contenu sera détruit. S'il n'existe pas, il sera créé. Notez que l'on obtiendrait un mode comparable à `w+` en ouvrant un fichier vide (mais existant) en mode `r+`.

**`a+`** : *extension et mise à jour*. Si le fichier n'existe pas, il sera créé. S'il existe, le pointeur sera positionné en fin de fichier.

**`t`** ou **`b`** : lorsque l'implémentation distingue les fichiers de texte des autres, il est possible d'ajouter l'une de ces deux lettres à chacun des 6 modes précédents. La lettre **`t`** précise que l'on a affaire à un fichier de texte ; la lettre **`b`** précise que l'on a affaire à un fichier binaire. (On dit aussi que **`t`** correspond au mode « translaté », pour spécifier que certaines substitutions auront lieu).

# 6 Les fichiers prédéfinis

Un certain nombre de fichiers sont connus du langage C, sans qu'il soit nécessaire ni de les ouvrir ni de les fermer :

- **stdin :** unité d'entrée (par défaut, le clavier) ;
- **stdout** : unité de sortie (par défaut, l'écran) ;
- **stderr :** unité d'affichage des messages d'erreurs (par défaut, l'écran).

On trouve parfois également :

- **stdaux** : unité auxiliaire ;
- **stdprt :** imprimante

Les deux premiers fichiers correspondent aux unités standard d'entrée et de sortie d'un programme. Lorsque vous exécutez un programme depuis le système, vous pouvez éventuellement rediriger ces fichiers. Par exemple, la commande système suivante (valable à la fois sous UNIX et sous DOS) :

```
TRUC    <DONNEES    >RESULTATS
```

exécute le programme `TRUC`, en utilisant comme unité d'entrée le fichier `DONNEES` et comme unité de sortie le fichier `RESULTATS`.

Dans ces conditions, une instruction telle que, par exemple, `fgetchar` deviendrait équivalente à `fgetc(fich)` où `fich` serait un flux obtenu par appel à `fopen`. De même, `scanf(...)` deviendrait équivalent à `fscanf(fich, ..)`, etc.

Notez bien qu'au sein du programme même il n'est pas possible de savoir si un fichier prédéfini a été redirigé au moment du lancement du programme ; autrement dit, lorsqu'une fonction comme `fgetchar` ou `scanf` lit des informations, elle ne peut absolument pas savoir si ces dernières proviennent du clavier ou d'un fichier.

**Remarque**

**Pour lire en toute tranquillité sur *stdin*.** Dans la section 2.3 du chapitre consacré aux chaînes de caractères, nous vous avons montré comment régler les problèmes posés par `scanf`, en faisant appel à l'association des deux fonctions `gets` et `sscanf`. Pour ce faire, nous avions dû toutefois supposer que les lignes lues par `gets` ne dépasseraient pas une certaine longueur. Cette hypothèse est déjà restrictive dans le cas d'informations provenant du clavier : même si cela peut paraître naturel à la plupart des utilisateurs de ne pas dépasser, par exemple, une largeur d'écran, le risque existe d'en voir certains entrer une ligne trop longue qui « plantera » le programme. Cette même hypothèse devient franchement intolérable dans le cas de lecture dans un fichier (sur lequel peut avoir été redirigée l'entrée standard !).

En fait, il est très simple de régler définitivement ce problème. Il suffit d'employer (revoyez l'exemple de la section 2.3 du chapitre consacré aux chaînes), à la place de :

```
gets (ligne) ;
```

une instruction telle que (`LG` désignant le nombre maximal de caractères acceptés) :

```
fgets (ligne, LG, stdin)
```

# Exercices

Tous ces exercices sont corrigés en fin de volume.

**1)** Écrire un programme permettant d'afficher le contenu d'un fichier texte en numérotant les lignes. Ces lignes ne devront jamais comporter plus de 80 caractères.

**2)** Écrire un programme permettant de créer séquentiellement un fichier « répertoire » comportant pour chaque personne :

- nom (20 caractères maximum) ;
- prénom (15 caractères maximum) ;
- âge (entier) ;
- numéro de téléphone (11 caractères maximum).

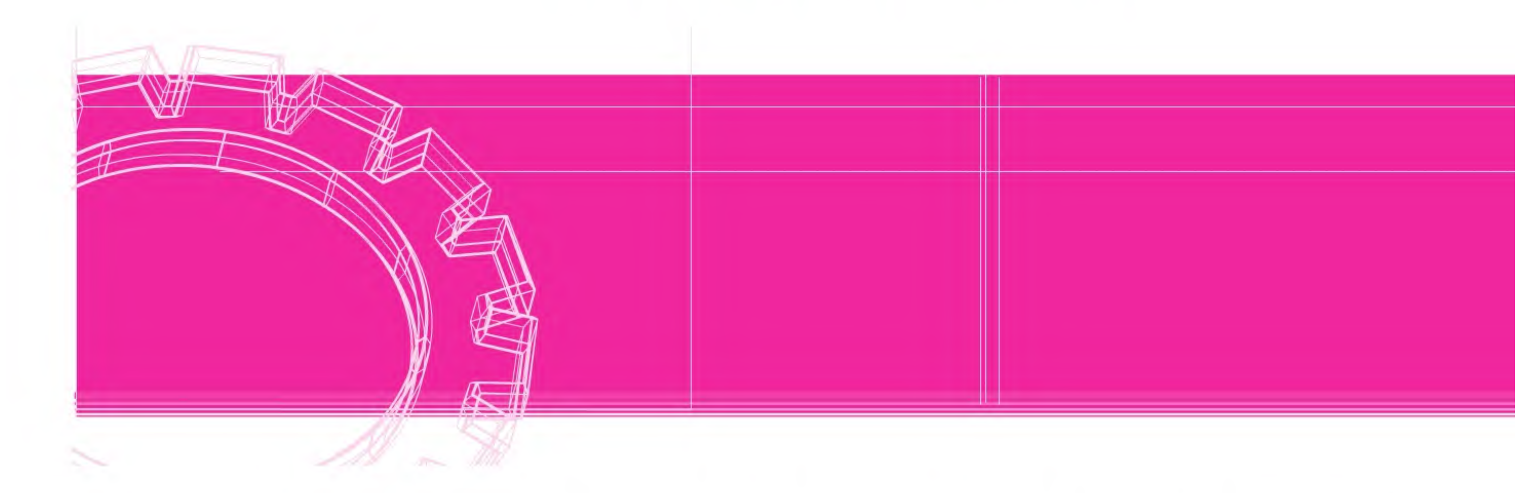
Les informations relatives aux différentes personnes seront lues au clavier.

**3)** Écrire un programme permettant, à partir du fichier créé par l'exercice précédent, de retrouver les informations correspondant à une personne de nom donné.

**4)** Écrire un programme permettant, à partir du fichier créé dans l'exercice 2, de retrouver les informations relatives à une personne de rang donné (par accès direct).

# Chapitre 11

# La gestion dynamique de la mémoire



Nous avons déjà eu l'occasion de faire la distinction entre les données statiques (variables globales ou locales statiques) et les données automatiques (variables locales). D'autre part, nous avons évoqué les possibilités d'allocation dynamique d'espace mémoire.

Cela signifie qu'en langage C un programme comporte en définitive trois types de données :

- statiques ;
- automatiques ;
- dynamiques.

Les **données statiques** occupent un emplacement parfaitement défini lors de la compilation.

Les **données automatiques**, en revanche, n'ont pas une taille définie a priori. En effet, elles ne sont créées et détruites qu'au fur et à mesure de l'exécution du programme. Elles sont souvent gérées sous forme de ce que l'on nomme une **pile** (*stack* en anglais), laquelle croît ou décroît suivant les besoins du programme. Plus précisément, elle croît à chaque entrée dans une fonction pour faire place à toutes les variables locales nécessaires pendant la durée de vie de la fonction ; elle décroît d'autant à chaque sortie.

Les **données dynamiques** n'ont pas non plus de taille définie a priori. Leur création ou leur libération dépend, cette fois, de demandes explicites faites lors de l'exécution du programme.

Leur gestion, qui ne saurait se faire à la manière d'une pile, est indépendante de celle des données automatiques. Plus précisément, elle se fait généralement dans ce que l'on nomme un **tas** (*heap* en anglais) dans lequel on cherche à allouer ou à libérer de l'espace en fonction des besoins.

En définitive, les données d'un programme se répartissent en trois catégories : statiques, automatiques et dynamiques. Les données statiques sont définies dès la compilation ; la gestion des données automatiques reste transparente au programmeur et seules les données dynamiques sont véritablement créées sur son initiative.

D'une manière générale, l'emploi de données statiques présente certains défauts intrinsèques. Citons deux exemples :

- Les données statiques ne permettent pas de définir des tableaux de dimensions variables, c'est-à-dire dont les dimensions peuvent être fixées lors de l'exécution et non dès la compilation. Il est alors nécessaire d'en fixer arbitrairement une taille limite, ce qui conduit généralement à une mauvaise utilisation de l'ensemble de la mémoire.
- La gestion statique ne se prête pas aisément à la mise en œuvre de listes chaînées, d'arbres binaires,... objets dont ni la structure ni l'ampleur ne sont généralement connues lors de la compilation du programme.

Les données dynamiques vont permettre de pallier ces défauts en donnant au programmeur l'opportunité de s'allouer et de libérer de la mémoire dans le « tas », au fur et à mesure de ses besoins.

# 1 Les outils de base de la gestion dynamique : `malloc` et `free`

Commençons par étudier les deux fonctions les plus classiques de gestion dynamique de la mémoire, à savoir `malloc` et `free`.

## 1.1 La fonction `malloc`

### a) Premier exemple

Considérez ces instructions :

```
#include <stdlib.h>
   .....
char * adr ;
   .....
adr = malloc (50) ;
   .....
for (i=0 ; i<50 ; i++) *(adr+i) = 'x' ;
```

L'appel :

```
malloc (50)
```

alloue un emplacement de 50 octets dans le tas et en fournit l'adresse en retour. Ici, cette dernière est placée dans le pointeur `adr`.

L'instruction `for` qui vient à la suite n'est donnée qu'à titre d'exemple d'utilisation de la zone ainsi créée.

### *b) Second exemple*

```
long * adr ;
   .....
adr = malloc (100 * sizeof(long)) ;
   .....
for (i=0 ; i<100 ; i++) *(adr+i) = 1 ;
```

Cette fois, nous nous sommes alloué une zone de `100 * sizeof(long)` octets (notez l'emploi de `sizeof` qui assure la portabilité). Mais nous l'avons considérée comme une suite de 100 entiers de type `long`. Pour ce faire, vous voyez que nous avons pris soin de placer le résultat de `malloc` dans un pointeur sur des éléments de type `long`. N'oubliez pas que les règles de l'arithmétique des pointeurs font que :

```
adr + i
```

correspond à l'adresse contenue dans `adr`, augmentée de `sizeof(long)` fois la valeur de `i` (puisque `adr` pointe sur des objets de longueur `sizeof(long)`).

### *c) D'une manière générale*

Le prototype de `malloc` (qui figure dans `stdlib.h`) est précisément :

```
void * malloc (size_t taille)          (stdlib.h)
```

Il montre tout d'abord que le résultat fourni par `malloc` est un pointeur générique (revoyez éventuellement le paragraphe correspondant du chapitre relatif aux pointeurs). Il pourra donc être converti implicitement en pointeur de n'importe quel type (c'est-à-dire sans qu'il soit nécessaire de faire appel explicitement à l'opérateur de `cast`) ; ainsi, dans nos précédents exemples, il a pu être converti en `char *` ou en `long *`. Une telle conversion peut apparaître relativement fictive, dans la mesure où l'adresse correspondante n'est pas modifiée par la conversion. Elle n'en a pas moins une grande importance puisque, comme nous l'avons déjà mentionné à diverses reprises, la connaissance du type d'un pointeur intervient dans les calculs arithmétiques portant sur ce pointeur.

D'autre part, nous constatons que l'unique argument de `malloc` est d'un type a priori inattendu (nous aurions pu penser à `int` ou `long`). En fait, `size_t` est un nom de type prédéfini par `typedef size_t` est précisément défini dans le fichier `stddef.h`, mais vous n'avez pas

besoin d'inclure vous-même ce fichier car cela est déjà prévu dans `stdlib.h`. Le type exact lui correspondant dépend de l'implémentation. Cela signifie donc que la taille maximale des objets que l'on peut s'allouer par `malloc` dépend de l'implémentation. (En pratique, toutefois, on peut toujours compter sur un minimum de 64 Ko).

Enfin, il faut savoir que `malloc` fournit un *pointeur nul* (`NULL`) dans le cas où l'allocation mémoire a échoué. Bien entendu, dans un programme opérationnel, il sera nécessaire de s'assurer qu'aucun problème de cette sorte n'apparaît.

## 1.2 La fonction free

L'un des intérêts essentiels de la gestion dynamique est de pouvoir récupérer des emplacements dont on n'a plus besoin. Le rôle de la fonction `free` est de libérer un emplacement préalablement alloué.

Voici un exemple de programme, exécuté ici dans un environnement DOS. Il vous montre comment `malloc` peut profiter d'un espace préalablement libéré sur le tas.

*Exemple d'utilisation de la fonction* free

```
#include <stdio.h>
#include <stdlib.h>
main()
{
   char * adr1, * adr2, * adr3 ;
   adr1 = malloc (100) ;
   printf ("allocation de 100 octets en %p\n", adr1) ;
   adr2 = malloc (50) ;
   printf ("allocation de  50 octets en %p\n", adr2) ;

   free (adr1) ;
   printf ("libération de 100 octets en %p\n", adr1) ;
   adr3 = malloc (40) ;
   printf ("allocation de  40 octets en %p\n", adr3) ;
}
```

```
allocation de 100 octets en 06AC
allocation de  50 octets en 0714
libération de 100 octets en 06AC
allocation de  40 octets en 06E8
```

Notez que la dernière allocation a pu se faire dans l'espace libéré par le précédent appel de `free`.

**Remarques**

Dans cet exemple, vous pouvez constater que l'allocation d'une zone de 100 octets nécessite en fait un peu plus de place mémoire (exactement 104 octets). La différence (4 octets) correspond à des « octets de service » dans lesquels le système place les informations nécessaires à sa gestion dynamique de la mémoire.

La norme prévoit que `malloc` alloue convenablement de l'espace, en tenant compte d'éventuelles contraintes d'alignement de l'objet concerné. Or, en toute rigueur, `malloc` n'a pas d'information précise sur le type de l'objet (elle n'en a que la longueur !). Dans ces conditions, le respect de la norme peut prendre des allures différentes suivant l'implémentation :

- alignement basé sur la taille demandée,
- alignement systématique tenant compte de la contrainte d'alignement la plus forte.

Dans tous les cas, bien sûr, aucun risque n'existe. Simplement, la taille mémoire réellement utilisée pourra, pour un type donné, différer d'une implémentation à une autre (notez qu'en toute rigueur c'est déjà ce qui se produit avec les octets de service dont le nombre peut varier avec l'implémentation).

# 2 D'autres outils de gestion dynamique : calloc et realloc

Bien qu'elles soient moins fondamentales que les précédentes, les deux fonctions `calloc` et `realloc` peuvent s'avérer pratiques dans certaines circonstances.

## 2.1 La fonction calloc

La fonction :

```
void * calloc ( size_t nb_blocs, size_t taille )     (stdlib.h)
```

alloue l'emplacement nécessaire à `nb_blocs` consécutifs, ayant chacun une taille de `taille` octets.

Contrairement à ce qui se passait avec `malloc`, où le contenu de l'espace mémoire alloué était imprévisible, `calloc` remet à zéro chacun des octets de la zone ainsi allouée.

La taille de chaque bloc, ainsi que leur nombre sont tous deux de type `size_t`. On voit ainsi qu'il est possible d'allouer en une seule fois une place mémoire (de plusieurs blocs) beaucoup plus importante que celle allouée par `malloc` (la taille limite théorique étant maintenant `size_t*size_t` au lieu de `size_t`).

**Remarques**

En général, l'allocation par calloc de p blocs de n octets conduira à utiliser un peu moins de mémoire que p allocations de n octets par malloc ; cela provient de ce que les éventuels octets de service ne seront attribués qu'une fois dans le premier cas (pour le système, il n'y a bien qu'une seule zone, même si nous avons formulé notre demande à malloc sous la forme de plusieurs blocs), alors qu'ils le seront p fois dans le second.

La remarque faite précédemment à propos des contraintes d'alignement s'applique également à calloc.

Une zone allouée par calloc ne peut être libérée qu'en une seule fois par free. Il n'est pas question d'essayer de n'en libérer qu'un morceau (comme nous l'avons déjà dit, les octets alloués par calloc forment un tout pour le système).

### 2.2 La fonction realloc

La fonction :

```
void * realloc (void * pointeur, size_t taille )      (stdlib.h)
```

permet de modifier la taille d'une zone préalablement allouée (par malloc, calloc ou realloc).

Le pointeur doit être l'adresse de début de la zone dont on veut modifier la taille. Quant à taille, de type size_t, elle représente la nouvelle taille souhaitée.

Cette fonction restitue l'adresse de la nouvelle zone ou un pointeur nul dans le cas où l'allocation a échoué.

Lorsque la nouvelle taille demandée est supérieure à l'ancienne, le contenu de l'ancienne zone est conservé (quitte à le recopier si la nouvelle adresse est différente de l'ancienne). Dans le cas où la nouvelle taille est inférieure à l'ancienne, le début de l'ancienne zone (c'est-à-dire taille octets) verra son contenu inchangé.

## 3 Exemple d'application de la gestion dynamique : création d'une liste chaînée

Comme nous l'avons déjà évoqué en introduction de ce chapitre, il n'est pas possible de déclarer un tableau dont le nombre d'éléments n'est pas connu lors de la compilation. En revanche, les possibilités de gestion dynamique du langage C nous permettent d'envisager d'allouer des emplacements aux différents éléments du tableau au fur et à mesure des besoins.



Si l'on n'a pas besoin d'accéder directement à chacun des éléments, on peut se contenter de constituer ce que l'on nomme une « liste chaînée », dans laquelle :

- un pointeur désigne le premier élément ;
- chaque élément comporte un pointeur sur l'élément suivant.

Dans ce cas, les emplacements des différents éléments peuvent être alloués de façon dynamique, au fur et à mesure des besoins. Il n'est plus nécessaire de connaître d'avance leur nombre ou une valeur maximale (ce qui serait le cas si l'on créait un tableau de tels éléments).

Appliquons cela à des éléments de type `point`, structure comportant les champs suivants :

```
        struct point { int num ;
                       float x ;
                       float y ;
                     } ;
```

Chaque élément doit donc contenir un pointeur sur un élément de même type. Il y a là une récursivité des déclarations qui est autorisée en C. Ainsi, nous pourrons adapter notre précédente structure de la manière suivante :

```
    struct element { int num ;
                     float x ;
                     float y ;
                     struct element * suivant ;
                   } ;
```

Vous voyez que nous avons été amené à utiliser dans la description du modèle `element` un pointeur sur ce même modèle.

Supposons que nous cherchions à constituer notre liste chaînée à partir d'informations fournies en données. Deux possibilités s'offrent à nous :

- ajouter chaque nouvel élément à la fin de la liste. Le parcours ultérieur de la liste se fera alors dans le même ordre que celui dans lequel les données ont été introduites.
- ajouter chaque nouvel élément au début de la liste. Le parcours ultérieur de la liste se fera alors dans l'ordre inverse de celui dans lequel les données ont été introduites.

Nous avons choisi ici de programmer la seconde méthode, laquelle se révèle légèrement plus simple que la deuxième.

Notez que le dernier élément de la liste (donc, dans notre cas, le premier lu) ne pointera sur rien. Or, lorsque nous chercherons ensuite à utiliser notre liste, il nous faudra être en mesure de *savoir où elle s'arrête*. Certes, nous pourrions, à cet effet, conserver l'adresse de son dernier élément. Mais il est plus simple d'attribuer au champ *suivant* de ce dernier élément une valeur fictive dont on sait qu'elle ne peut apparaître par ailleurs. La valeur `NULL` (`0`) fait très bien l'affaire.

Ici, nous avons décidé de faire effectuer la création de la liste par une fonction. Le programme principal se contente de réserver l'emplacement d'un pointeur destiné à désigner le premier élément de la liste. Sa valeur effective sera fournie par la fonction `creation`. Dans ces conditions, il est nécessaire que le programme principal lui fournisse, non pas la valeur, mais l'adresse de ce pointeur (du moins si l'on souhaite pouvoir disposer ultérieurement de cette valeur au sein du programme principal).

C'est ce qui justifie la forme de l'en-tête de la fonction `creation` :

```
void creation (struct element * * adeb)
```

dans laquelle `adeb` est effectivement du type « pointeur sur un pointeur sur un élément de type `struct element` ».

*Création d'une liste chaînée*

```
#include <stdio.h>
#include <stdlib.h>
struct element { int num ;
                 float x ;
                 float y ;
                 struct element * suivant ;
               } ;
void creation (struct element * * adeb) ;
main()
{
   struct element *  debut ;
   creation (&debut) ;
}
void creation (struct element * * adeb)
{
   int num ;
   float x, y ;
   struct element * courant ;
   * adeb = NULL ;
   while ( printf("numéro x y : "),
           scanf ("%d %f %f", &num, &x, &y), num)
       { courant = (struct element *) malloc (sizeof(struct element)) ;
         courant -> num     = num ;
         courant -> x       = x ;
         courant -> y       = y ;
         courant -> suivant = * adeb ;
         * adeb = courant ;
       }
}
```

Bien entendu, la constitution d'une telle liste n'est souvent que le préalable à un traitement plus sophistiqué. En particulier, dans un cas réel, on pourrait être amené à réaliser des opérations telles que :

- insertion d'un nouvel élément dans la liste ;
- suppression d'un élément de la liste.

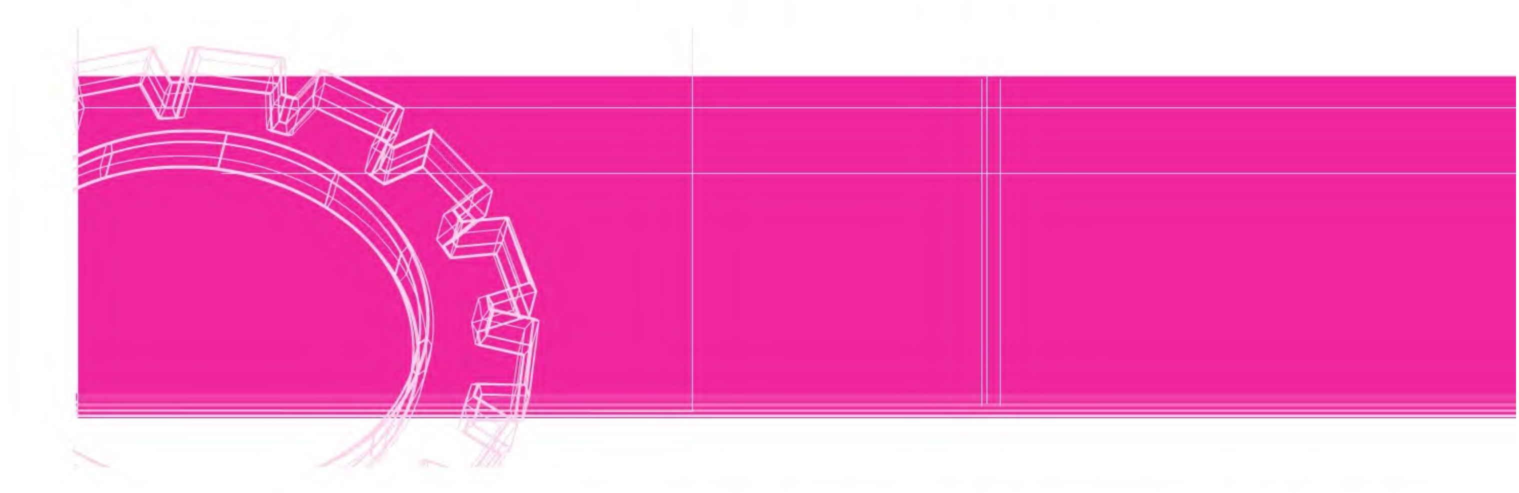
# Exercice

Cet exercice est corrigé en fin de volume.

Ajouter au programme de création d'une liste chaînée du paragraphe 3 une fonction permettant d'afficher le contenu de la liste chaînée précédemment créée. Cette fonction possédera comme unique argument l'adresse de début de la liste.

# Chapitre 12

# Le préprocesseur



Nous avons déjà été amené à évoquer l'existence d'un « préprocesseur ». Il s'agit d'un programme qui est exécuté automatiquement avant la compilation et qui transforme votre fichier source à partir d'un certain nombre de directives. Ces dernières, contrairement à ce qui se produit pour les instructions du langage C, sont écrites sur des lignes distinctes du reste du programme ; elles sont toujours introduites par un mot précis commençant par le caractère #.

Parmi ces directives, nous avons déjà utilisé `#include` et `#define`. Nous nous proposons ici d'étudier les diverses possibilités offertes par le préprocesseur, à savoir :

- l'incorporation de fichiers source (directive `#include`) ;
- la définition de symboles et de macros (directive `#define`) ;
- la compilation conditionnelle.



## 1 La directive `#include`

Elle permet d'incorporer, avant compilation, le texte figurant dans un fichier quelconque. Jusqu'ici, nous n'avions incorporé de cette manière que les contenus de fichiers en-tête requis pour le bon usage des fonctions standard. Mais cette directive peut s'appliquer également à des fichiers de votre cru. Cela peut s'avérer utile, par exemple pour écrire une seule fois les

déclarations communes à plusieurs fichiers source différents, en particulier dans le cas des prototypes de fonctions.

Rappelons que cette directive possède deux syntaxes voisines :

```
#include <nom_fichier>
```

recherche le fichier mentionné dans un emplacement (chemin, répertoire) défini par l'implémentation.

```
#include "nom_fichier"
```

recherche le fichier mentionné dans le même emplacement (chemin, répertoire) que celui où se trouve le programme source.

Généralement, la première est utilisée pour les fichiers en-tête correspondant à la bibliothèque standard, tandis que la seconde l'est plutôt pour les fichiers que vous créez vous-même.

**Remarques**

Un fichier incorporé par `#include` peut lui-même comporter, à son tour, des directives `#include` ; c'est d'ailleurs le cas de certains fichiers en-tête relatifs à la bibliothèque standard. D'une manière générale, le nombre maximal de niveaux d'imbrication des directives `#include` dépend de l'implémentation ; toutefois, en pratique, il n'est jamais inférieur à 5.

Lorsque vous créez vos propres fichiers en-tête, il vous faut prendre garde à ne pas introduire plusieurs fois des déclarations identiques, ce qui risquerait de conduire à des erreurs de compilation. Ce point est particulièrement crucial dans le cas d'imbrication des directives `#include`. D'une manière générale, ce genre de problème, qui se pose d'ailleurs fréquemment avec les fichiers en-tête standard, se résout par l'emploi de directives conditionnelles, associé à la définition de symboles particuliers (nous y reviendrons dans le paragraphe 3).

# 2 La directive #define

Elle offre en fait deux possibilités :

- définition de symboles (c'est sous cette forme que nous l'avons employée jusqu'ici) ;
- définition de macros.

## 2.1 Définition de symboles

Une directive telle que :

```
        #define  nbmax   5
```

demande de substituer au symbole `nbmax` le texte `5`, et cela chaque fois que ce symbole apparaîtra dans la suite du fichier source.

Une directive :

```
#define  entier  int
```

placée en début de programme, permettra d'écrire en français les déclarations de variables entières. Ainsi, par exemple, ces instructions :

```
entier a, b ;
entier * p ;
```

seront remplacées par :

```
int a, b ;
int * p ;
```

Il est possible de demander de faire apparaître dans le texte de substitution un symbole déjà défini. Par exemple, avec ces directives :

```
#define  nbmax  5
 ....
#define  taille  nbmax + 1
```

Chaque mot `taille` apparaissant dans la suite du programme sera systématiquement remplacé par `5+1`. Notez bien que `taille` ne sera pas remplacé exactement par `6` mais, étant donné que le compilateur accepte les expressions constantes là où les constantes sont autorisées, le résultat sera comparable (après compilation).

Il est même possible de demander de substituer à un symbole un texte vide. Par exemple, avec cette directive :

```
#define  rien
```

tous les symboles `rien` figurant dans la suite du programme seront remplacés par un texte vide. Tout se passera donc comme s'ils ne figuraient pas dans le programme.

Nous verrons qu'une telle possibilité n'est pas aussi fantaisiste qu'il y paraît au premier abord puisqu'elle pourra intervenir dans la compilation conditionnelle.

Voici quelques derniers exemples vous montrant comment résumer en un seul mot une instruction C :

```
#define  bonjour  printf("bonjour")
#define  affiche  printf("resultat %d\n", a)
#define  ligne    printf("\n")
```

Notez que nous aurions pu inclure le point-virgule de fin dans le texte de substitution.

D'une manière générale, la syntaxe de cette directive fait que le symbole à remplacer ne peut contenir d'espace (puisque le premier espace sert de délimiteur entre le symbole à substituer et le texte de substitution). Le texte de substitution, quant à lui, peut contenir autant d'espaces

que vous le souhaitez puisque c'est la fin de ligne qui termine la directive. Il est même possible de le prolonger au-delà, en terminant la ligne par \ et en poursuivant sur la ligne suivante.

Vous voyez que, compte tenu des possibilités d'imbrication des substitutions, cette instruction s'avère très puissante au point qu'elle pourrait permettre à celui qui le souhaiterait de réécrire totalement le langage C (on pourrait, par exemple, le franciser). Cette puissance a toutefois ses propres limites dans la mesure où tout abus dans ce sens conduit inexorablement à une perte de lisibilité des programmes.

**Remarques**

Si vous introduisez, par mégarde, un signe = dans une directive `#define`, aucune erreur ne sera, bien sûr, détectée par le préprocesseur lui-même. Par contre, en général, cela conduira à une erreur de compilation. Ainsi, par exemple, avec :

```
#define N = 5
```

une instruction telle que :

```
int t[N] ;
```

deviendra, après traitement par le préprocesseur :

```
int t[= 5] ;
```

laquelle est manifestement erronée. Notez bien, toutefois, que, la plupart du temps, vous ne connaîtrez pas le texte généré par le préprocesseur et vous serez simplement en présence d'un diagnostic de compilation concernant apparemment l'instruction `int t[N]`. Le diagnostic de l'erreur en sera d'autant plus délicat.

Une autre erreur aussi courante que la précédente consiste à terminer (à tort) une directive `#include` par un point-virgule. Les considérations précédentes restent valables dans ce cas.

Certaines implémentations permettent d'avoir connaissance du texte généré par le préprocesseur, c'est-à-dire, du texte qui sera véritablement compilé ; cette facilité peut rendre plus aisé le diagnostic d'erreurs telles que celles que nous venons d'envisager.

## 2.2 Définition de macros

La définition de macros ressemble à la définition de symboles mais elle fait intervenir la notion de paramètres.

Par exemple, avec cette directive :

```
#define  carre(a)  a*a
```

le préprocesseur remplacera dans la suite tous les textes de la forme :

```
carre(x)
```

dans lesquels x représente en fait un symbole quelconque par :

```
x*x
```

Par exemple :

```
carre(z)         deviendra   z*z
carre(valeur)    deviendra   valeur*valeur
carre(12)        deviendra   12*12
```

La macro précédente ne disposait que d'un seul paramètre, mais il est possible d'en faire intervenir plusieurs en les séparant, classiquement, par des virgules. Par exemple, avec :

```
      #define  dif(a,b)  a-b
```

```
dif(x,z)           deviendrait   x-z
dif(valeur+9,n)    deviendrait   valeur+9-n
```

Là encore, les définitions peuvent s'imbriquer. Ainsi, avec les deux définitions précédentes, le texte :

```
dif(carre(p),carre(q))
```

sera, dans un premier temps, remplacé par :

```
dif(p*p,q*q)
```

puis, dans un second temps, par :

```
p*p-q*q
```

Néanmoins, malgré la puissance de cette directive, il ne faut pas oublier que, dans tous les cas, il ne s'agit que de substitution de texte. Il est souvent nécessaire de prendre quelques précautions, notamment lorsque le texte de substitution fait intervenir des opérateurs. Par exemple, avec ces instructions :

```
#define DOUBLE(x) x + x
     .....
DOUBLE(a)/b
DOUBLE(x+2*y)
DOUBLE(x++)
```

Le texte généré par le préprocesseur sera le suivant :

```
a + a/b
x+2*y + x+2*y
x++ + x++
```

Vous constatez que, si le premier appel de macro conduit à un résultat correct, le deuxième ne fournit pas, comme on aurait pu l'escompter, le double de l'expression figurant en paramètre. Quant au troisième, il fait apparaître ce que l'on nomme souvent un « effet de bord ». En effet, la notation :

```
DOUBLE(x++)
```

conduit à incrémenter deux fois la variable `x`. De plus, elle ne fournit pas vraiment son double. Par exemple, si `x` contient la valeur `5`, l'exécution du programme ainsi généré conduira à calculer `5+6`.

Le premier problème, lié aux priorités relatives des opérateurs, peut être facilement résolu en introduisant des parenthèses dans la définition de la macro. Ainsi, avec :

```
#define DOUBLE(x)  ((x)+(x))
   ...
DOUBLE(a)/b
DOUBLE(x+2*y)
DOUBLE(x++)
```

Le texte généré par le préprocesseur sera :

```
((a)+(a))/b
((x+2*y)+(x+2*y))
((x++)+(x++))
```

Les choses sont nettement plus satisfaisantes pour les deux premiers appels de la macro `DOUBLE`. Par contre, bien entendu, l'effet de bord introduit par le troisième n'a pas pour autant disparu.

Par ailleurs, il faut savoir que les substitutions de paramètres ne se font pas à l'intérieur des chaînes de caractères. Ainsi, avec ces instructions :

```
#define AFFICHE(y) printf("valeur de y %d",y)
   ...
AFFICHE(a) ;
AFFICHE(c+5) ;
```

le texte généré par le préprocesseur sera :

```
printf("valeur de y %d",a) ;
printf("valeur de y %d",c+5) ;
```

**Remarque**

Dans la définition d'une macro, il est impératif de ne pas prévoir d'espace dans la partie spécifiant le nom de la macro et les différents paramètres. En effet, là encore, le premier espace sert à délimiter la macro à définir. Par exemple, avec :

```
#define somme (a,b) a+b
   ...
z = somme(x+5) ;
```

le préprocesseur générerait le texte :

```
z = (a,b) a+b(x+5) ;
```

**Remarque C99** La norme C99 permet de définir des « fonctions en ligne ». Elles sont repérées par le mot *inline* figurant devant leur en-tête et leurs instructions sont incorporées à chaque appel, comme le sont celles des macros. Mais les fonctions en ligne présentent l'avantage sur les macros de ne plus induire d'effets de bords.

# 3 La compilation conditionnelle

Un certain nombre de directives permettent d'incorporer ou d'exclure des portions du fichier source dans le texte qui est analysé par le préprocesseur. Ces directives se classent en deux catégories en fonction de la condition qui régit l'incorporation :

- existence ou inexistence de symboles ;
- valeur d'une expression.

## 3.1 Incorporation liée à l'existence de symboles

```
#ifdef symbole
  .....
#else
  .....
#endif
```

demande d'incorporer le texte figurant entre les deux lignes `#ifdef` et `#else` si le symbole indiqué est effectivement défini au moment où l'on rencontre `#ifdef`. Dans le cas contraire, c'est le texte figurant entre `#else` et `#endif` qui sera incorporé. La directive `#else` peut, naturellement, être absente.

De façon comparable :

```
#ifndef symbole
 .....
#else
 .....
#endif
```

demande d'incorporer le texte figurant entre les deux lignes #ifndef et #else si le symbole indiqué n'est pas défini. Dans le cas contraire, c'est le texte figurant entre #else et #endif qui sera incorporé.

Notez bien que, pour qu'un tel symbole soit effectivement défini pour le préprocesseur, il doit faire l'objet d'une directive #define. Notamment, ne confondez pas ces symboles avec d'éventuelles variables qui pourraient être déclarées par des instructions C classiques, et qui, quant à elles, ne sont absolument pas connues du préprocesseur.

Voici un exemple d'utilisation de ces directives :

```
#define MISEAUPOINT
   .....
#ifdef MISEAUPOINT
   instructions 1
#else
   instructions 2
#endif
```

Ici, les instructions 1 seront incorporées par le préprocesseur, tandis que les instructions 2 ne le seront pas. En revanche, il suffirait de supprimer la directive #define MISEAUPOINT pour aboutir au résultat contraire.

**Remarque**

Ces définitions de symboles sont fréquemment utilisées dans les fichiers en-tête standard. Ils permettent notamment d'inclure, depuis un fichier en-tête donné, un autre fichier en-tête, en s'assurant que ce dernier n'a pas déjà été inclus (afin d'éviter la duplication de certaines instructions risquant de conduire à des erreurs de compilation). La même technique peut, bien sûr, s'appliquer à vos propres fichiers en-tête.

## 3.2 Incorporation liée à la valeur d'une expression

La construction ci-après :

```
#if condition
 .....
#else
 .....
#endif
```

permet d'incorporer l'une des deux parties du texte, suivant la valeur de la condition indiquée.

En voici un exemple d'utilisation :

```
#define CODE 1
   .....
#if CODE == 1
   instructions 1
#endif
#if CODE == 2
   instructions 2
#endif
```

Ici, ce sont les instructions 1 qui seront incorporées par le préprocesseur. Mais il s'agirait des instructions 2 si nous remplacions la première directive par :

```
#define CODE 2
```

Notez qu'il existe également une directive #elif qui permet de condenser les choix imbriqués. Par exemple, nos précédentes instructions pourraient s'écrire :

```
#define CODE 1
   .....
#if CODE == 1
   instructions 1
#elif CODE == 2
   instructions 2
#endif
```

D'une manière générale, la condition mentionnée dans ces directives #if et #elif peut faire intervenir n'importe quels symboles définis pour le préprocesseur et des opérateurs relationnels, arithmétiques ou logiques. Ces derniers se notent exactement de la même manière qu'en langage C.

En outre, il existe un opérateur noté defined, utilisable uniquement dans les conditions destinées au préprocesseur (if et elif). Ainsi, l'exemple donné à la fin de la section 3.1 pourrait également s'écrire :

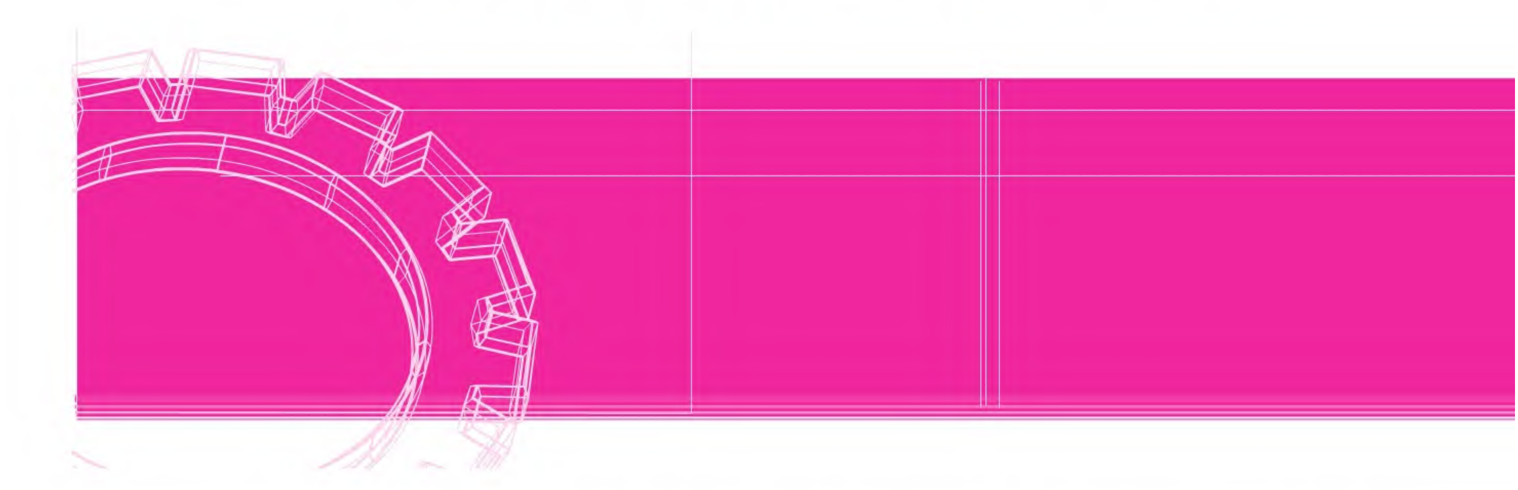
```
#define MISEAUPOINT
   .....
#if defined(MISEAUPOINT)
   instructions 1
#else
   instructions 2
#endif
```

D'une manière générale, les directives de test de la valeur d'une expression peuvent s'avérer précieuses :

- pour introduire dans un fichier source des instructions de mise au point que l'on pourra ainsi introduire ou supprimer à volonté du module objet correspondant. Par une intervention mineure au niveau du source lui-même, il est possible de contrôler la présence ou l'absence de ces instructions dans le module objet correspondant, et ainsi, de ne pas le pénaliser en taille mémoire lorsque le programme est au point.
- pour adapter un programme unique à différents environnements. Les paramètres définissant l'environnement sont alors exprimés dans des symboles du préprocesseur.

# Chapitre 13

# Les possibilités du langage C proches de la machine



L'une des caractéristiques du langage C est précisément d'offrir des possibilités de programmation comparables à celles de l'assembleur (ou du langage machine). Ce chapitre se propose de vous les présenter. Après avoir effectué quelques rappels sur le codage des nombres en binaire, nous apporterons quelques précisions sur le « qualificatif de signe » (`signed/unsigned`) que nous avions éludé jusqu'ici et nous verrons comment, dans ce cas, se généralisent les règles de conversion. Nous étudierons ensuite les opérateurs de manipulation de bits puis nous verrons comment, dans une structure, définir des champs ayant une taille inférieure à un octet (à l'aide de « champs de bits »). Enfin, nous aborderons ce que l'on nomme les « unions ».

D'une manière générale, sachez que tous les points évoqués dans ce chapitre sont, par essence même, peu portables. Leur emploi devra généralement soit être limité à des programmes destinés à un matériel donné, soit être parfaitement « localisé » au sein du programme, afin de permettre, le cas échéant, l'adaptation du programme à une autre machine.

# 1 Compléments sur les types d'entiers

## 1.1 Rappels concernant la représentation des nombres entiers en binaire

Pour fixer les idées, nous raisonnerons ici sur des nombres entiers représentés sur 16 bits, mais il va de soi qu'il serait facile de généraliser notre propos à une taille quelconque.

Quelle que soit la machine (et donc, a fortiori, le langage !), les entiers sont codés en utilisant un bit pour représenter le signe (0 pour positif et 1 pour négatif).

**a)** Lorsqu'il s'agit d'un nombre **positif** (ou nul), sa valeur absolue est écrite en base 2, à la suite du bit de signe. Voici quelques exemples de codages de nombres (à gauche, le nombre en décimal, au centre, le codage binaire correspondant, à droite, le même codage exprimé en hexadécimal) :

| | | |
|---|---|---|
| 1 | 0000000000000001 | 0001 |
| 2 | 0000000000000010 | 0002 |
| 3 | 0000000000000011 | 0003 |
| 16 | 0000000000010000 | 0010 |
| 127 | 0000000001111111 | 007F |
| 255 | 0000000011111111 | 00FF |

**b)** Lorsqu'il s'agit d'un nombre **négatif**, sa valeur absolue est alors codée suivant ce que l'on nomme la « technique du complément à deux ». Pour ce faire, cette valeur est d'abord exprimée en base 2 puis tous les bits sont inversés (1 devient 0 et 0 devient 1) et, enfin, on ajoute une unité au résultat. Voici quelques exemples (avec la même présentation que précédemment) :

| | | |
|---|---|---|
| -1 | 1111111111111111 | FFFF |
| -2 | 1111111111111110 | FFFE |
| -3 | 1111111111111101 | FFFD |
| -4 | 1111111111111100 | FFFC |
| -16 | 1111111111110000 | FFF0 |
| -256 | 1111111100000000 | FF00 |

**Remarques**

Le nombre 0 est codé d'une seule manière (0000000000000000).

Si l'on ajoute 1 au plus grand nombre positif (ici 0111111111111111, soit 7FFF en hexadécimal ou 32768 en décimal) et que l'on ne tient pas compte de la dernière retenue (ou, ce qui revient au même, si l'on ne considère que les 16 derniers bits du résultat), on obtient... le plus petit nombre négatif possible (ici 1000000000000000, soit 8000 en hexadécimal ou -32768 en décimal). C'est ce qui explique le phénomène de modulo bien connu de l'arithmétique entière (les dépassements de capacité n'étant jamais signalés, quel que soit le langage considéré).

## 1.2 Prise en compte d'un attribut de signe

Ce que nous venons d'exposer s'applique, en C, aux types `short`, `int` et `long` (avec différents nombres de bits). Mais le langage C vous autorise à définir trois autres types voisins des précédents en utilisant le qualificatif `unsigned`. Dans ce cas, on ne représente plus que des nombres positifs pour lesquels aucun bit de signe n'est nécessaire. Cela permet de doubler la taille des nombres représentables ; par exemple, avec 16 bits, on passe de l'intervalle [-32768; 32767] à l'intervalle [0 ; 65535].

Toutefois, il ne faut pas perdre de vue que la notion de type n'est pas intrinsèque, autrement dit, lorsqu'on voit un motif binaire donné, on ne peut pas lui attribuer de valeur précise, tant qu'on ne sait pas comment il a été codé. Précisément, à un entier de 16 bits on pourra attribuer deux valeurs différentes, suivant qu'on le considère comme « signé » ou « non signé ». Par exemple, `1111111111111111` vaut `-1` quand on le considère comme signé et `65535` quand on le considère comme non signé.

D'une manière similaire, `avec printf ("%u", n)` on obtiendra (n étant le même dans les deux cas) une valeur différente de celle fournie par `printf ("%d", n)` ; en effet, rappelons que la fonction `printf` n'a plus connaissance du type exact des valeurs qu'elle reçoit et que seul le code de format lui permet d'effectuer le « bon décodage ».

## 1.3 Extension des règles de conversions

Tant qu'une expression ne mélange pas des types signés et des types non signés, les choses restent naturelles. Il suffit simplement de compléter les conversions `short -> int -> long` par les conversions `unsigned short -> unsigned int -> unsigned long`, mais aucun problème nouveau ne se pose (on peut, certes, toujours obtenir des dépassements de capacité qui ne seront pas détectés).

En revanche, le mélange des types signés et des types non signés est (hélas !) accepté par le langage ; mais il conduit à mettre en place des conversions (généralement de signé vers non signé) n'ayant guère de sens et ne respectant pas, de toute façon, l'intégrité des données (que pourrait d'ailleurs bien valoir `-5` converti en non signé ?). Une telle liberté est donc à proscrire. À simple titre indicatif, sachez que les conversions prévues par la norme, dans ce cas, se contentent de préserver le motif binaire (par exemple, la conversion de `signed int` en `unsigned int` revient à conserver tel quel le motif binaire concerné).

La même remarque prévaut pour les conversions forcées par une affectation ou par un opérateur de « cast ».

## 1.4 La notation octale ou hexadécimale des constantes

Pour écrire une constante entière, vous pouvez utiliser une notation octale (base 8) ou hexadécimale (base 16). La forme octale se note en faisant précéder le nombre écrit en base 8 du chiffre 0.

Par exemple :

`014` correspond à la valeur décimale `12`,

`037` correspond à la valeur décimale `31`.

La forme hexadécimale se note en faisant précéder le nombre écrit en hexadécimal (les dix premiers chiffres se notent 0 à 9, A correspond à dix, B à onze,... F à quinze) des deux caractères `0x` (ou `0X`). Par exemple :

`0x1A` correspond à la valeur décimale `26` (`16+10`)

Les deux dernières notations doivent cependant être réservées aux situations dans lesquelles on s'intéresse plus au motif binaire qu'à la valeur numérique de la constante en question. D'ailleurs, tout se passe comme si l'on avait affaire, dans ce cas, à une valeur non signée et, là encore, il sera préférable d'éviter tout mélange (dans une même expression) avec des valeurs signées.

# 2 Compléments sur les types de caractères

## 2.1 Prise en compte d'un attribut de signe

A priori, le type `char` peut, lui aussi, recevoir un attribut de signe ; en outre, la norme ne dit pas si `char` tout court correspond à `unsigned char` ou à `signed char` (alors que, par défaut, tous les types entiers sont considérés comme signés).

L'attribut de signe d'une variable de type caractère n'a aucune incidence sur la manière dont un caractère donné est représenté (codé) : il n'y a qu'un seul jeu de codes sur 8 bits, soit 256 combinaisons possibles en comptant le `\0`. En revanche, cet attribut va intervenir dès lors qu'on s'intéresse à la valeur numérique associée au caractère et non plus au caractère lui-même. C'est le cas, par exemple, dans des expressions telles que les suivantes (`c` étant supposé de type caractère) :

```
c + 1
c++
printf ("%d", c) ;
```

Pour fixer les idées, supposons, ici encore, que les entiers de type `int` sont représentés sur 16 bits et voyons comment se déroule précisément la conversion `char -> int` en question.

**a)** Si le caractère n'est pas signé, on ajoute à gauche de son code 8 bits égaux à 0. Par exemple :

`01001110` devient `0000000001001110`

**b)** Si le caractère est signé, on ajoute à gauche de son code 8 bits égaux au premier bit du code du caractère. Par exemple :

`01001110` devient `0000000001001110`
`11011001` devient `1111111111011001`

Cette démarche revient, en fait, à considérer que les 8 bits du code du caractère représentent un (petit) entier codé lui aussi suivant la technique du complément à deux, avec bit de signe à gauche. L'opération de conversion alors décrite correspond à ce que l'on nomme la « propagation du bit de signe » ; elle permet d'obtenir un nombre entier sur 16 bits identique à celui qui était représenté sur 8 bits. On peut dire que, vu comme cela, l'intégrité des données est préservée, exactement comme elle l'était dans une conversion telle que `int -> long` (qui, au demeurant, emploie exactement la même technique de propagation du bit de signe).

Ainsi, si `n` est de type `int` et que `c` est de type caractère, l'instruction :

```
n = c ;
```

conduira à affecter à `n` :

- une valeur comprise entre `-128` et `127` si `c` est de type `unsigned char` ;
- une valeur comprise entre `0` et `255` si `c` est de type `signed char`.

La même remarque s'applique à la valeur affichée par :

```
printf ("%d", c) ;
```

## 2.2 Extension des règles de conversion

Notez qu'il n'y a aucune conversion d'un type caractère vers un autre type caractère, compte tenu des conversions systématiques. En revanche, les conversions d'un entier vers un caractère sont autorisées (dans les affectations ou avec l'opérateur de « cast »). Dans ce cas, elles sont simplement mises en œuvre en ne conservant de l'entier que les bits les moins significatifs : le motif binaire obtenu est le même, que le caractère en question soit signé ou non signé.

Théoriquement, de telles conversions apparaissent dans de banales affectations telles que :

```
c = c + 1 ;        /* ou c++ ; */
```

En pratique, et quel que soit l'attribut de signe de `c`, compte tenu de ce qui vient d'être dit, tout se passe finalement comme si l'on avait ajouté `1` à 8 bits, sans tenir compte d'un éventuel dépassement de capacité (et d'ailleurs, il n'est pas dit que certains compilateurs ne fassent pas cela directement, sans passer par des conversions entier -> caractère -> entier).

# 3 Les opérateurs de manipulation de bits

## 3.1 Présentation des opérateurs de manipulation de bits

Le langage C dispose d'opérateurs permettant de travailler directement sur le motif binaire d'une valeur. Ceux-ci lui procurent ainsi des possibilités traditionnellement réservées à la programmation en langage assembleur.

A priori, **ces opérateurs ne peuvent porter que sur des types entiers**. Toutefois, compte tenu des règles de conversion implicite, ils pourront également s'appliquer à des caractères (mais le résultat en sera entier).

Le tableau ci-après fournit la liste de ces opérateurs qui se composent de cinq opérateurs binaires et d'un opérateur unaire.

*Opérateurs de manipulation de bits*

| TYPE | OPÉRATEUR | SIGNIFICATION |
|---|---|---|
| binaire | & | ET (bit à bit) |
| | \| | OU inclusif (bit à bit) |
| | ^ | OU exclusif (bit à bit) |
| | << | Décalage à gauche |
| | >> | Décalage à droite |
| unaire | ~ | Complément à un (bit à bit) |

## 3.2 Les opérateurs bit à bit

Les 3 opérateurs &, | et ^ appliquent en fait la même opération à chacun des bits des deux opérandes. Leur résultat peut ainsi être défini à partir d'une table (dite « table de vérité ») fournissant le résultat de cette opération lorsqu'on la fait porter sur deux bits de même rang de chacun des deux opérandes. Cette table est fournie par le tableau ci-après.

L'opérateur unaire ~ (dit de « complément à un ») est également du type « bit à bit ». Il se contente d'inverser chacun des bits de son unique opérande (0 donne 1 et 1 donne 0).

*Table de vérité des opérateurs « bit à bit »*

| OPÉRANDE 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|
| OPÉRANDE 2 | 0 | 1 | 0 | 1 |
| ET (&) | 0 | 0 | 0 | 1 |
| OU inclusif (\|) | 0 | 1 | 1 | 1 |
| OU exclusif (^) | 0 | 1 | 1 | 0 |



Voici quelques exemples de résultats obtenus à l'aide de ces opérateurs. Nous avons supposé que les variables n et p étaient toutes deux du type int et que ce dernier utilisait 16 bits. Nous avons systématiquement indiqué les valeurs sous forme binaire, hexadécimale et décimale :

| | | | |
|---|---|---|---|
| n | 0000010101101110 | 056E | 1390 |
| p | 0000001110110011 | 03B3 | 947 |
| n & p | 0000000100100010 | 0122 | 290 |
| n \| p | 0000011111111111 | 07FF | 2047 |
| n ^ p | 0000011011011101 | 06DD | 1757 |
| ~ n | 1111101010010001 | FA91 | -1391 |

Notez que le qualificatif signed/unsigned des opérandes n'a pas d'incidence sur le motif binaire créé par ces opérateurs. Cependant, le type même du résultat produit se trouve défini par les règles habituelles. Ainsi, dans nos précédents exemples, la valeur décimale de ~n serait 64145 si n avait été déclaré unsigned int (ce qui ne change pas son motif binaire).

## 3.3 Les opérateurs de décalage

Ils permettent de réaliser des décalages à droite ou à gauche sur le motif binaire correspondant à leur premier opérande. L'amplitude du décalage, exprimée en nombre de bits, est fournie par le second opérande. Par exemple :

```
n << 2
```

fournit comme résultat la valeur obtenue en décalant le motif binaire de n de 2 bits vers la gauche ; les bits de gauche sont perdus et des bits à zéro apparaissent à droite.

De même :

```
n >> 3
```

fournit comme résultat la valeur obtenue en décalant le motif binaire de n de 3 bits vers la droite. Cette fois, les bits de droite sont perdus, tandis que des bits apparaissent à gauche.

Ces derniers dépendent du qualificatif signed/unsigned du premier opérande. S'il s'agit de unsigned, les bits ainsi créés à gauche sont à zéro. S'il s'agit de signed, les bits ainsi créés seront égaux au bit signe (il y a, ici encore, propagation du bit signe).

Voici quelques exemples de résultats obtenus à l'aide de ces opérateurs de décalage. La variable n est supposée signed int, tandis que p est supposée unsigned int.

| | | |
|---|---|---|
| (signed) n | 0000010101101110 | 1010110111011110 |
| (unsigned) p | 0000010101101110 | 1010110111011110 |
| n << 2 | 0001010110111000 | 1011011101111000 |
| n >> 3 | 0000000010101101 | 1111011011101111 |
| p >> 3 | 0000000010101101 | 0001010110111011 |

### 3.4 Exemples d'utilisation des opérateurs de bits

L'opérateur `&` permet d'accéder à une partie des bits d'une valeur en masquant les autres. Par exemple, l'expression :

```
n & 0xF
```

permet de ne prendre en compte que les 4 bits de droite de `n` (que `n` soit de type `char`, `short`, `int` ou `long`).

De même :

```
n & 0x8000
```

permet d'extraire le « bit signe » de `n`, supposé de type `int` dans une implémentation où celui-ci correspond à 16 bits.

Voici un exemple de programme qui décide si un entier est pair ou impair, en examinant simplement le dernier bit de sa représentation binaire :

*Test de la parité d'un entier*

```
#include <stdio.h>
main()
{
   int n ;
   printf ("donnez un entier : ") ;
   scanf ("%d", &n) ;
   if ( n & 1 == 1 )
         printf ("il est impair") ;
      else
         printf ("il est pair") ;
}
```

```
donnez un entier : 58
il est pair
_________________
donnez un entier : 47
il est impair
```

## 4 Les champs de bits

Nous venons de voir que le langage C dispose d'opérateurs de bits très puissants permettant de travailler au niveau du bit. De plus, ce langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.

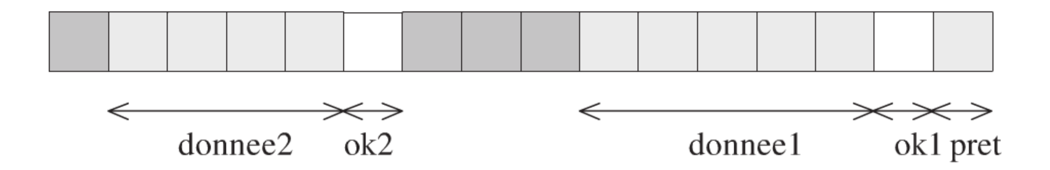
Cela peut s'avérer utile :

- soit pour compacter l'information : par exemple, un nombre entier compris entre 0 et 15 pourra être rangé sur 4 bits au lieu de 16 (encore faudra-t-il utiliser convenablement les bits restants) ;
- soit pour décortiquer le contenu d'un motif binaire, par exemple un mot d'état en provenance d'un périphérique spécialisé.

Voyez cet exemple de déclaration :

```
struct etat
     { unsigned pret : 1 ;
       unsigned ok1  : 1 ;
       int donnee1   : 5 ;
       int           : 3 ;
       unsigned ok2  : 1 ;
       int donnee2   : 4 ;
     } ;
struct etat mot ;
```

La variable `mot` ainsi déclarée peut être schématisée comme suit :




Les indications figurant à la suite des « deux-points » précisent la longueur du champ en bits. Lorsque aucun nom de champ ne figure devant cette indication de longueur, cela signifie que l'on saute le nombre de bits correspondants (ils ne seront donc pas utilisés).

Avec ces déclarations, la notation :

```
mot.donnee1
```

désigne un entier signé pouvant prendre des valeurs comprises entre `-16` et `+15`. Elle pourra apparaître à n'importe quel endroit où C autorise l'emploi d'une variable de type `int`.

Les seuls types susceptibles d'apparaître dans des champs de bits sont **int** et **unsigned int**. Notez que lorsqu'un champ de type `int` est de longueur 1, ses valeurs possibles sont `0` et `-1` (et non `0` et `1`, comme ce serait le cas avec le type `unsigned int`).

**Remarques**

La norme ne précise pas si la description d'un champ de bits se fait en allant des poids faibles vers les poids forts ou dans le sens inverse. Ce point dépend donc de l'implémentation et, en pratique, on rencontre les deux situations (y compris pour différents compilateurs sur une même machine !). En outre, lorsqu'un champ de bits occupe plusieurs octets, l'ordre dans lequel ces derniers sont décrits dépend, lui aussi, de l'implémentation.

La taille maximale d'un champ de bits dépend, elle aussi, de l'implémentation. En pratique, on rencontre fréquemment 16 bits ou 32 bits.

L'emploi des champs de bits est, donc, par nature même, peu ou pas portable. Il doit, par conséquent, être réservé à des applications très spécifiques.

# 5 Les unions

L'union, en langage C, permet de faire partager un même emplacement mémoire par des variables de types différents. Cela peut s'avérer utile :

- pour économiser des emplacements mémoire, en utilisant un même emplacement pendant des phases différentes d'un même programme ; d'une manière générale, vous verrez que les possibilités de gestion dynamique du langage C résolvent ce problème d'une façon plus pratique ;
- pour interpréter de plusieurs façons différentes un même motif binaire. Dans ce cas, il sera alors fréquent que l'union soit elle-même associée à des champs de bits.

Voyez d'abord cet exemple introductif qui n'a d'intérêt que dans une implémentation dans laquelle les types `float` et `long` ont la même taille :

*Union entre un entier et un flottant (supposés de même taille)*

```
#include <stdio.h>
main()
{
  union essai
         { long n ;
           float x ;
         }  u ;
   printf ("donnez un nombre réel : ")   ;
   scanf ("%f", &u.x) ;
   printf (" en entier, cela fait : %ld", u.n) ;
}
```

```
donnez un nombre réel : 1.23e4
en entier, cela fait : 1178611712
```



La déclaration :

```
union essai
      { long n ;
        float x ;
      }  u ;
```

réserve un emplacement dont le nombre de bits correspond à la taille (ici supposée commune) d'un `long` ou d'un `float` qui pourra être considéré tantôt comme un entier long qu'on désignera alors par **u.n**, tantôt comme un flottant (`float`) qu'on désignera alors par **u.x**.

D'une manière générale, la syntaxe de la description d'une union est analogue à celle d'une structure. Elle possède un nom de modèle (ici `essai`, nous aurions d'ailleurs pu l'omettre) ; celui-ci peut être ensuite utilisé pour définir d'autres variables de ce type. Par exemple, dans notre précédent programme, nous pourrions déclarer d'autres objets du même type que `u` par :

```
union essai z, truc ;
```

Par ailleurs, il est possible de réaliser une union portant sur plus de deux objets ; d'autre part, chaque objet peut être non seulement d'un type de base (comme dans notre exemple), mais également de type structure. En voici un exemple dans lequel nous réalisons une union entre une structure `etat` telle que nous l'avions définie dans le paragraphe précédent et un entier (cela n'aura d'intérêt que dans des implémentations où le type `int` occupe 16 bits).

```
struct etat
     { unsigned pret : 1 ;
       unsigned ok1  : 1 ;
       int donnee1   : 5 ;
       int           : 3 ;
       unsigned ok2  : 1 ;
       int donnee2   : 4 ;
     } ;
union
     { int valeur ;
       struct etat bits ;
     } mot ;
```

Notez qu'ici nous n'avons pas donné de nom au modèle d'union et nous y avons déclaré directement une variable `mot`.

Avec ces déclarations, il est alors possible, par exemple, d'accéder à la valeur de `mot`, considéré comme un entier, en la désignant par :

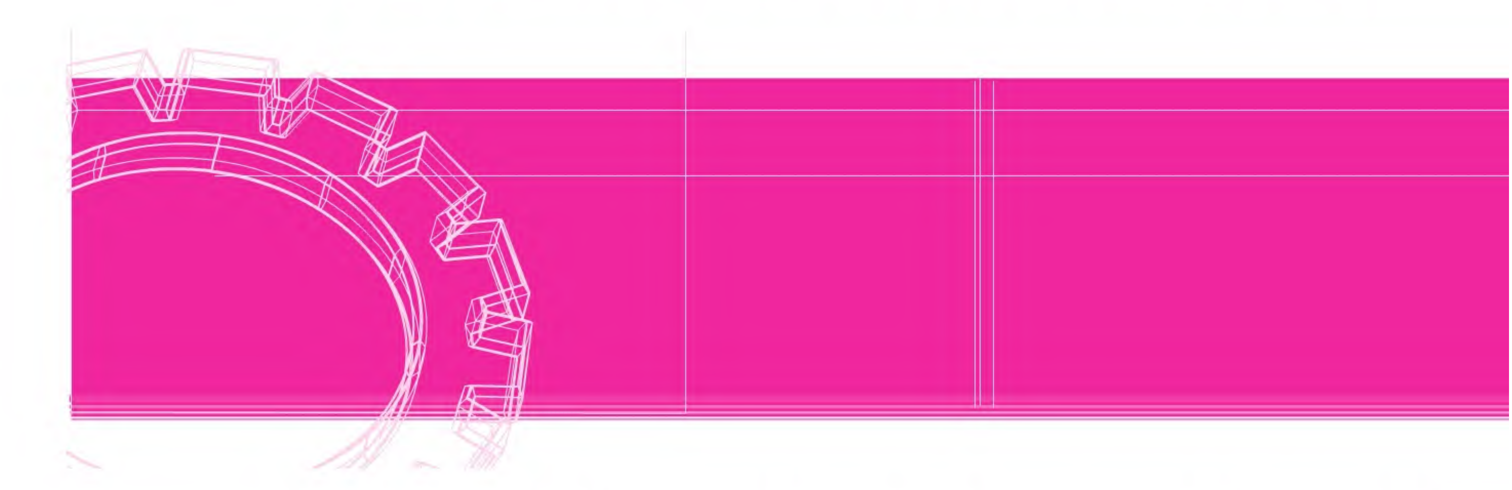
```
mot.valeur
```

Quant aux différentes parties désignant ce mot, il sera possible d'y accéder en les désignant par :

```
mot.bits.pret
mot.bits.ok1
mot.bits.donnee1
etc.
```

# Annexe

# Les principales fonctions de la bibliothèque standard



Comme nous l'avons déjà mentionné, la norme ANSI fournit à la fois la description du langage C et le contenu d'une bibliothèque standard. Plus précisément, cette bibliothèque est subdivisée en plusieurs sous-bibliothèques ; à chaque sous-bibliothèque est associé un fichier « en-tête » (d'extension `.h`) comportant essentiellement :

- les en-têtes des fonctions correspondantes ;
- les définitions des macros correspondantes ;
- les définitions de certains symboles utiles au bon fonctionnement des fonctions ou macros de la sous-bibliothèque.

La présente annexe décrit les **principales fonctions** prévues par la norme. Chaque paragraphe correspond à une sous-bibliothèque et précise quel est le nom du fichier en-tête correspondant.

**Remarque**

Les fonctions décrites ici sont classées par fichier en-tête, et non par ordre alphabétique. Néanmoins, si vous cherchez la description d'une fonction précise, il vous suffit de vous reporter à l'index situé en fin d'ouvrage.

# 1 Entrées-sorties (stdio.h)

## 1.1 Gestion des fichiers

**FOPEN** **FILE * fopen (const char * nomfichier, const char * mode)**

Ouvre le fichier dont le nom est fourni, sous forme d'une chaîne, à l'adresse indiquée par `nomfichier`. Fournit, en retour, un « flux » (pointeur sur une structure de type prédéfini `FILE`), ou un pointeur nul si l'ouverture a échoué. Les valeurs possibles de `mode` sont décrites dans le chapitre traitant des fichiers.

**FCLOSE** **int fclose (FILE * flux)**

Vide éventuellement le tampon associé au flux concerné, désalloue l'espace mémoire attribué à ce tampon et ferme le fichier correspondant. Fournit la valeur `EOF` en cas d'erreur et la valeur `0` dans le cas contraire.

## 1.2 Écriture formatée

Toutes ces fonctions utilisent une chaîne de caractères nommée *format*, composée à la fois de caractères quelconques et de codes de format dont la signification est décrite en détail à la fin du présent paragraphe.

**FPRINTF** **int fprintf (FILE * flux, const char * format, ...)**

Convertit les valeurs éventuellement mentionnées dans la liste d'arguments (...) en fonction du `format` spécifié, puis écrit le résultat dans le `flux` indiqué. Fournit le nombre de caractères effectivement écrits ou une valeur négative en cas d'erreur.

**PRINTF** **int printf (const char * format, ...)**

Convertit les valeurs éventuellement mentionnées dans la liste d'arguments (...) en fonction du `format` spécifié, puis écrit le résultat sur la sortie standard (`stdout`). Fournit le nombre de caractères effectivement écrits ou une valeur négative en cas d'erreur.

Notez que :

```
printf (format, ...) ;
```

est équivalent à :

```
fprintf (stdout, format, ...) ;
```



**SPRINTF** **int sprintf (char * ch, const char * format, ...)**

Convertit les valeurs éventuellement mentionnées dans la liste d'arguments (...) en fonction du `format` spécifié et place le résultat dans la chaîne d'adresse `ch`, en le complétant par un caractère `\0`. Fournit le nombre de caractères effectivement écrits (sans tenir compte du `\0`) ou une valeur négative en cas d'erreur.

## Les codes de format utilisables avec ces trois fonctions

Chaque code de format a la structure suivante :

**% [drapeaux] [largeur] [.précision] [h|l|L] conversion**

dans laquelle les crochets `[` et `]` signifient que ce qu'ils renferment est facultatif. Les différentes « indications » se définissent comme suit :

- **drapeaux :**

**-** : justification à gauche

**+** : signe toujours présent

**^** : impression d'un espace au lieu du signe +

**#** : forme alternée ; elle n'affecte que les types **o**, **x**, **X**, **e**, **E**, **f**, **g** et **G** comme suit :

- **o** : fait précéder de `0` toute valeur non nulle
- **x** ou **X** : fait précéder de 0x ou 0X la valeur affichée
- **e**, **E** ou **f** : le point décimal apparaît toujours
- **g** ou **G** : même effet que pour **e** ou **E**, mais de plus les zéros de droite ne seront pas supprimés

- **largeur** (`n` désigne une constante entière positive écrite en notation décimale) :

**n** : au minimum, `n` caractères seront affichés, éventuellement complétés par des blancs à gauche

**0n** : au minimum, `n` caractères seront affichés, éventuellement complétés par des zéros à gauche

***** : la largeur effective est fournie dans la liste d'expressions

- **précision** (n désigne une constante entière positive écrite en notation décimale) :

**.n** : la signification dépend du caractère de conversion, de la manière suivante :

- **d**, **i**, **o**, **u**, **x** ou **X** : au moins n chiffres seront imprimés. Si le nombre comporte moins de n chiffres, l'affichage sera complété à gauche par des zéros. Notez que cela n'est pas contradictoire avec l'indication de largeur, si celle-ci est supérieure à n. En effet, dans ce cas, le nombre pourra être précédé à la fois d'espaces et de zéros
- **e**, **E** ou **f** : on obtiendra n chiffres après le point décimal, avec arrondi du dernier
- **g** ou **G** : on obtiendra au maximum n chiffres significatifs
- **c** : sans effet
- **s** : au maximum n caractères seront affichés. Notez que cela n'est pas contradictoire avec l'indication de largeur

**.0** : la signification dépend du caractère de conversion, comme suit :

- **d**, **i**, **o**, **u**, **x** ou **X** : choix de la valeur par défaut de la précision (voir ci-dessous)
- **e**, **E** ou **f** : pas d'affichage du point décimal

***** : la valeur effective de n est fournie dans la « liste d'expressions »

**rien** : choix de la valeur par défaut, à savoir :

- 1 pour d, i, o, u, x ou X
- 6 pour e, E ou f
- tous les chiffres significatifs pour g ou G
- tous les caractères pour s
- sans effet pour c

- **h|l|L** :

**h** : l'expression correspondante est d'un type short int (signé ou non). En fait, il faut voir que, compte tenu des conversions implicites, printf ne peut jamais recevoir de valeur d'un tel type. Tout au plus peut-elle recevoir un entier dont on (le programmeur) sait qu'il résulte de la conversion d'un short. Dans certaines implémentations, l'emploi du modificateur h conduit alors à afficher la valeur correspondante suivant un gabarit différent de celui réservé à un int (c'est souvent le cas pour le nombre de caractères hexadécimaux). Ce code ne peut, de toute façon, avoir une éventuelle signification que pour les caractères de conversion : d, i, o, u, x ou X

**l** : Ce code précise que l'expression correspondante est de type long int. Il n'a de signification que pour les caractères de conversion : d, i, o, u, x ou X

**L** : Ce code précise que l'expression correspondante est de type long double. Il n'a de signification que pour les caractères de conversion : e, E, f, g ou G

- **conversion** : il s'agit d'un caractère qui précise à la fois le type de l'expression (nous l'avons noté ici en italique) et la façon de présenter sa valeur. Les types numériques indiqués correspondent au cas où aucun modificateur n'est utilisé (voir ci-dessus) :
  - **d** : `signed int`, affiché en décimal
  - **o** : `unsigned int`, affiché en octal
  - **u** : `unsigned int`, affiché en décimal
  - **x** : `unsigned int`, affiché en hexadécimal (lettres minuscules)
  - **X** : `signed int`, affiché en hexadécimal (lettres majuscules)
  - **f** : `double`, affiché en notation décimale
  - **e** : `double`, affiché en notation exponentielle (avec la lettre `e`)
  - **E** : `double`, affiché en notation exponentielle (avec la lettre `E`)
  - **g** : `double`, affiché suivant le code `f` ou `e` (ce dernier étant utilisé lorsque l'exposant obtenu est soit supérieur à la précision désirée, soit inférieur à `-4`)
  - **G** : `double`, affiché suivant le code `f` ou `E` (ce dernier étant utilisé lorsque l'exposant obtenu est soit supérieur à la précision désirée, soit inférieur à `-4`)
  - **c** : `char`
  - **s** : pointeur sur une « chaîne »
  - **%** : affiche le caractère `%`, sans faire appel à aucune expression de la liste
  - **n** : place, à l'adresse désignée par l'expression de la liste (du type pointeur sur un entier), le nombre de caractères écrits jusqu'ici
  - **p** : pointeur, affiché sous une forme dépendant de l'implémentation

## 1.3 Lecture formatée

Ces fonctions utilisent une chaîne de caractères nommée `format`, composée à la fois de caractères quelconques et de codes de format dont la signification est décrite en détail à la fin du présent paragraphe. On y trouvera également les règles générales auxquelles obéissent ces fonctions (arrêt du traitement d'un code de format, arrêt prématuré de la fonction).

**FSCANF** **`int fscanf (FILE * flux, const char * format, ...)`**

Lit des caractères sur le flux spécifié, les convertit en tenant compte du `format` indiqué et affecte les valeurs obtenues aux différentes variables de la liste d'arguments (...). Fournit le nombre de valeurs lues convenablement ou la valeur `EOF` si une erreur s'est produite ou si une fin de fichier a été rencontrée avant qu'une seule valeur ait pu être lue.

**SCANF** **int scanf (const char * format, ...)**

Lit des caractères sur l'entrée standard (`stdin`), les convertit en tenant compte du `format` indiqué et affecte les valeurs obtenues aux différentes variables de la liste d'arguments (...). Fournit le nombre de valeurs lues convenablement ou la valeur `EOF` si une erreur s'est produite ou si une fin de fichier a été rencontrée avant qu'une seule valeur ait pu être lue.

Notez que :

```
scanf (format, ...)
```

est équivalent à :

```
fscanf (stdin, format, ...)
```

**SSCANF** **int sscanf (char * ch, const char * format, ...)**

Lit des caractères dans la chaîne d'adresse `ch`, les convertit en tenant compte du `format` indiqué et affecte les valeurs obtenues aux différentes variables de la liste d'arguments (...). Fournit le nombre de valeurs lues convenablement.

## Règles communes à ces fonctions

**a)** Il existe six caractères dits « séparateurs », à savoir : l'espace, la tabulation horizontale (`\t`), la fin de ligne (`\n`), le retour chariot (`\r`), la tabulation verticale (`\v`) et le changement de page (`\f`). En pratique, on se limite généralement à l'espace et à la fin de ligne.

**b)** L'information est recherchée dans un tampon, image d'une ligne. Il y a donc une certaine désynchronisation entre ce que l'on frappe au clavier (lorsque l'unité standard est connectée à ce périphérique) et ce que lit *la fonction*. Lorsqu'il n'y a plus d'information disponible dans le tampon, il y a déclenchement de la lecture d'une nouvelle ligne. Pour décrire l'exploration de ce tampon, il est plus simple de faire intervenir un indicateur de position que nous nommerons pointeur.

**c)** La rencontre dans le format d'un caractère séparateur provoque l'avancement du pointeur jusqu'à la rencontre d'un caractère qui ne soit pas un séparateur.

**d)** La rencontre dans le format d'un caractère différent d'un séparateur (et de `%`) provoque la prise en compte du caractère courant (celui désigné par le pointeur). Si celui-ci correspond au caractère du format, *la fonction* poursuit son exploration du format. Dans le cas contraire, il y a arrêt prématuré de *la fonction*.

**e)** Lors du traitement d'un code de format, l'exploration s'arrête :

- à la rencontre d'un caractère invalide par rapport à l'usage qu'on doit en faire (point décimal pour un entier, caractère différent d'un chiffre ou d'un signe pour du numérique,...). Si *la fonction* n'est pas en mesure de fabriquer une valeur, il y a arrêt prématuré de l'ensemble de la lecture,
- à la rencontre d'un séparateur,
- lorsque la longueur (si elle a été spécifiée) a été atteinte.

## Les codes de format utilisés par ces fonctions

Chaque code de format a la structure suivante :

```
% [*] [largeur] [h|l|L] conversion
```

dans laquelle les crochets `[` et `]` signifient que ce qu'ils renferment est facultatif. Les différentes « indications » se définissent comme suit :

- **`*`** : la valeur lue n'est pas prise en compte ; elle n'est donc affectée à aucun élément de la liste
- **largeur** : nombre maximal de caractères à prendre en compte (on peut en lire moins s'il y a rencontre d'un séparateur ou d'un caractère invalide)
- **h|l|L** :
  - **`h`** : l'élément correspondant est l'adresse d'un `short int`. Ce modificateur n'a de signification que pour les caractères de conversion : `d`, `i`, `n`, `o`, `u`, ou `x`
  - **`l`** : l'élément correspondant est l'adresse d'un élément de type :
    - `long int` pour les caractères de conversion `d`, `i`, `n`, `o`, `u` ou `x`
    - `double` pour les caractères de conversion `e` ou `f`
  - **`L`** : l'élément correspondant est l'adresse d'un élément de type `long double`. Ce modificateur n'a de signification que pour les caractères de conversion `e`, `f` ou `g`.
- conversion : ce caractère précise à la fois le type de l'élément correspondant (nous l'avons indiqué ici en italique) et la manière dont sa valeur sera exprimée. Les types numériques indiqués correspondent au cas où aucun modificateur n'est utilisé (voir ci-dessus). Il ne faut pas perdre de vue que l'élément correspondant est toujours désigné par son adresse. Ainsi, par exemple, lorsque nous parlons de `signed int`, il faut lire : « adresse d'un `signed int` » ou encore « pointeur sur un `signed int` ».
  - **`d`** : `signed int` exprimé en décimal
  - **`o`** : `signed int` exprimé en octal
  - **`i`** : `signed int` exprimé en décimal, en octal ou en hexadécimal
  - **`u`** : `unsigned int` exprimé en décimal
  - **`x`** : `int` (`signed` ou `unsigned`) exprimé en hexadécimal

- **f**, **e** ou **g** : `float` écrit indifféremment en notation décimale (éventuellement sans point) ou exponentielle (avec e ou E)
- **c** : suivant la longueur, correspond à :
  - un *caractère* lorsqu'aucune longueur n'est spécifiée ou que celle-ci est égale à `1`
  - une *suite de caractères* lorsqu'une longueur différente de `1` est spécifiée. Dans ce cas, il ne faut pas perdre de vue que la fonction reçoit une adresse et que donc, dans ce cas, elle lira le nombre de caractères spécifiés et les rangera à partir de l'adresse indiquée. Il est bien sûr préférable que la place nécessaire ait été réservée. Notez bien qu'il ne s'agit pas ici d'une véritable chaîne, puisqu'il n'y aura pas (à l'image de ce qui se passe pour le code `%s`) d'introduction du caractère `\0` à la suite des caractères rangés en mémoire
- **s** : *chaîne de caractères*. Il ne faut pas perdre de vue que la fonction reçoit une adresse et que donc, dans ce cas, elle lira tous les caractères jusqu'à la rencontre d'un séparateur (ou un nombre de caractères égal à la longueur éventuellement spécifiée) et elle les rangera à partir de l'adresse indiquée. Il est donc préférable que la place nécessaire ait été réservée. Notez bien qu'ici un caractère `\0` est stocké à la suite des caractères rangés en mémoire et que sa place aura dû être prévue (si l'on lit `n` caractères, il faudra de la place sur `n+1`)
- **n** : `int`, dans lequel sera placé le nombre de caractères lus correctement jusqu'ici. Aucun caractère n'est donc lu par cette spécification
- **p** : *pointeur* exprimé en hexadécimal, sous la forme employée par `printf` (elle dépend de l'implémentation)

## 1.4 Entrées-sorties de caractères

**FGETC** **`int fgetc (FILE * flux)`**

Lit le caractère courant du `flux` indiqué. Fournit :

- le résultat de la conversion en `int` du caractère `c` (considéré comme `unsigned int`) si l'on n'était pas en fin de fichier
- la valeur `EOF` si la fin de fichier était atteinte

Notez que `fgetc` ne fournit de valeur négative qu'en cas de fin de fichier, quel que soit le code employé pour représenter les caractères et quel que soit l'attribut de signe attribué par défaut au type `char`.



**FGETS** **char * fgets (char * ch, int n, FILE * flux)**

Lit au maximum `n-1` caractères sur le `flux` mentionné (en s'interrompant éventuellement en cas de rencontre d'un caractère `\n`), les range dans la chaîne d'adresse `ch`, puis complète le tout par un caractère `\0`. Le caractère `\n`, s'il a été lu, est lui aussi rangé dans la chaîne (donc juste avant le `\0`). Cette fonction fournit en retour :

- la valeur `NULL` si une éventuelle erreur a eu lieu ou si une fin de fichier a été rencontrée,
- l'adresse `ch`, dans le cas contraire.

**FPUTC** **int fputc (int c, FILE * flux)**

Écrit sur le `flux` mentionné la valeur de `c`, après conversion en `unsigned char`. Fournit la valeur du caractère écrit (qui peut donc, éventuellement, être différente de celle du caractère reçu) ou la valeur `EOF` en cas d'erreur.

**FPUTS** **int fputs (const char * ch, FILE * flux)**

Écrit la chaîne d'adresse `ch` sur le `flux` mentionné. Fournit la valeur `EOF` en cas d'erreur et une valeur non négative dans le cas contraire.

**GETC** **int getc (FILE * flux)**

Macro effectuant la même chose que la fonction `fgetc`.

**GETCHAR** **int getchar (void)**

Macro effectuant la même chose que l'appel de la macro :

```
fgetc (stdin)
```

**GETS** **char * gets (char * ch)**

Lit des caractères sur l'entrée standard (`stdin`), en s'interrompant à la rencontre d'une fin de ligne (`\n`) ou d'une fin de fichier, et les range dans la chaîne d'adresse `ch`, en remplaçant le `\n` par `\0`. Fournit :

- la valeur `NULL` si une erreur a eu lieu ou si une fin de fichier a été rencontrée, alors qu'aucun caractère n'a encore été lu,
- l'adresse `ch`, dans le cas contraire.

**PUTC** **int putc (int c, FILE * flux)**

Macro effectuant la même chose que la fonction `fputc`.

**PUTCHAR** **int putchar (int c)**

Macro effectuant la même chose que l'appel de la macro `putc`, avec `stdout` comme adresse de flux. Ainsi :

```
putchar (c)
```

est équivalent à :

```
putc (c, stdout)
```

**PUTS** **int puts (const char * ch)**

Écrit sur l'unité standard de sortie (`stdout`) la chaîne d'adresse `ch`, suivie d'une fin de ligne (`\n`). Fournit `EOF` en cas d'erreur et une valeur non négative dans le cas contraire.

**Remarque**

On n'est pas sûr de pouvoir effectuer plusieurs appels consécutifs de `ungetc`, sans lectures intermédiaires.

## 1.5 Entrées-sorties sans formatage

**FREAD** **size_t fread (void * adr, size_t taille, size_t nblocs, FILE * flux)**

Lit, sur le `flux` spécifié, au maximum `nblocs` de `taille` octets chacun et les range à l'adresse `adr`. Fournit le nombre de blocs réellement lus.

**FWRITE** **size_t fwrite (const void * adr, size_t taille, size_t nblocs, FILE * flux)**

Écrit, sur le `flux` spécifié, `nblocs` de `taille` octets chacun, à partir de l'adresse `adr`. Fournit le nombre de blocs réellement écrits.

## 1.6 Action sur le pointeur de fichier

**FSEEK** **int fseek (FILE * flux, long noct, int org)**

Place le pointeur du `flux` indiqué à un endroit défini comme étant situé à `noct` octets de l'« origine » spécifiée par `org` :

`org = SEEK_SET` correspond au début du fichier

`org = SEEK_CUR` correspond à la position actuelle du pointeur

`org = SEEK_END` correspond à la fin du fichier

Dans le cas des fichiers de texte (si l'implémentation les différencie des autres), les seules possibilités autorisées sont l'une des deux suivantes :

- `noct = 0`
- `noct` a la valeur fournie par `ftell` (voir ci-dessous) et `org = SEEK_SET`

**FTELL** `long ftell (FILE *flux)`

Fournit la position courante du pointeur du `flux` indiqué (exprimée en octets par rapport au début du fichier) ou la valeur `-1L` en cas d'erreur.

### 1.7 Gestion des erreurs

**FEOF** `int feof (FILE * flux)`

Fournit une valeur non nulle si l'indicateur de fin de fichier du `flux` indiqué est activé et la valeur `0` dans le cas contraire.

## 2 Tests de caractères et conversions majuscules-minuscules (ctype.h)

**ISALNUM** `int isalnum (char c)`

Fournit la valeur `1` (vrai) si `c` est une lettre ou un chiffre et la valeur `0` (faux) dans le cas contraire.

**ISALPHA** `int isalpha (char c)`

Fournit la valeur `1` (vrai) si `c` est une lettre et la valeur `0` (faux) dans le cas contraire.

**ISDIGIT** `int isdigit (char c)`

Fournit la valeur `1` (vrai) si `c` est un chiffre et la valeur `0` (faux) dans le cas contraire.

**ISLOWER** `int islower (char c)`

Fournit la valeur `1` (vrai) si `c` est une lettre minuscule et la valeur `0` (faux) dans le cas contraire.

**ISSPACE** `int isspace (char c)`

Fournit la valeur 1 (vrai) si `c` est un séparateur (espace, saut de page, fin de ligne, tabulation horizontale ou verticale) et la valeur 0 (faux) dans le cas contraire.

**ISUPPER** `int isupper (char c)`

Fournit la valeur `1` (vrai) si `c` est une lettre majuscule et la valeur `0` (faux) dans le cas contraire.

# 3 Manipulation de chaînes (string.h)

**STRCPY** **char * strcpy (char * but, const char * source)**

Copie la chaîne source à l'adresse but (y compris le \0 de fin) et fournit en retour l'adresse de but.

**STRNCPY** **char * strncpy (char * but, const char * source, int lgmax)**

Copie au maximum lgmax caractères de la chaîne source à l'adresse but en complétant éventuellement par des caractères \0 si cette longueur maximale n'est pas atteinte. Fournit en retour l'adresse de but.

**STRCAT** **char * strcat (char * but, const char * source)**

Recopie la chaîne source à la fin de la chaîne but et fournit en retour l'adresse de but.

**STRNCAT** **char * strncat (char * but, const char * source, size_t lgmax)**

Recopie au maximum lgmax caractères de la chaîne source à la fin de la chaîne but et fournit en retour l'adresse de but.

**STRCMP** **int strcmp (const char * chaine1, const char * chaine2)**

Compare chaine1 et chaine2 et fournit :

- une valeur négative si chaine1 < chaine2 ;
- une valeur positive si chaine1 > chaine2 ;
- zéro si chaine1 = chaine2.

**STRNCMP** **int strncmp (const char * chaine1, const char * chaine2, size_t lgmax)**

Travaille comme strcmp, en limitant la comparaison à un maximum de lgmax caractères.

**STRCHR** **char * strchr (const char * chaine, char c)**

Fournit un pointeur sur la première occurrence du caractère c dans la chaîne chaine, ou un pointeur nul si ce caractère n'y figure pas.

**STRRCHR** **char * strrchr (const char * chaine, char c)**

Fournit un pointeur sur la dernière occurrence du caractère c dans la chaîne chaine ou un pointeur nul si ce caractère n'y figure pas.



**STRSPN** **size_t strspn (const char * chaine1, const char * chaine2)**

Fournit la longueur du segment initial de `chaine1` formé entièrement de caractères appartenant à `chaine2`.

**STRCSPN** **size_t strcspn (const char * chaine1, const char * chaine2)**

Fournit la longueur du segment initial de `chaine1` formé entièrement de caractères n'appartenant pas à `chaine2`.

**STRSTR** **char * strstr (const char * chaine1, const char * chaine2)**

Fournit un pointeur sur la première occurrence dans `chaine1` de `chaine2` ou un pointeur nul si `chaine2` ne figure pas dans `chaine1`.

**STRLEN** **size_t strlen (const char * chaine)**

Fournit la longueur de `chaine`.

**MEMCPY** **void * memcpy (void * but, const void * source, size_t lg)**

Copie `lg` octets depuis l'adresse `source` à l'adresse `but` qu'elle fournit comme valeur de retour (il ne doit **pas** y avoir **de recoupement** entre `source` et `but`).

**MEMMOVE** **void * memmove (void * but, const void * source, size_t lg)**

Copie `lg` octets depuis l'adresse `source` à l'adresse `but` qu'elle fournit comme valeur de retour (il peut y avoir **recoupement** entre `source` et `but`).

## 4 Fonctions mathématiques (math.h)

**SIN** **double sin (double x)**

**COS** **double cos (double x)**

**TAN** **double tan (double x)**

**ASIN** **double asin (double x)**

**ACOS** **double acos (double x)**

**ATAN** **double atan (double x)**

**ATAN2** **double atan2 (double y, double x)**

Fournit la valeur de arctan($y/x$)

| | | |
|---|---|---|
| **SINH** | **`double sinh (double x)`** | |
| | Fournit la valeur de sh($x$) | |

**SINH** — **`double sinh (double x)`**

Fournit la valeur de sh($x$)

**COSH** — **`double cosh (double x)`**

Fournit la valeur de ch($x$)

**TANH** — **`double tanh (double x)`**

Fournit la valeur de th($x$)

**EXP** — **`double exp (double x)`**

**LOG** — **`double log (double x)`**

Fournit la valeur du logarithme népérien de $x$ : Ln($x$) (ou Log($x$))

**LOG10** — **`double log10 (double x)`**

Fournit la valeur du logarithme à base 10 de $x$ : log($x$)

**POW** — **`double pow (double x, double y)`**

Fournit la valeur de $x^y$

**SQRT** — **`double sqrt (double x)`**

**CEIL** — **`double ceil (double x)`**

Fournit (sous forme d'un `double`) le plus petit entier qui ne soit pas inférieur à $x$.

**FLOOR** — **`double floor (double x)`**

Fournit (sous forme d'un `double`) le plus grand entier qui ne soit pas supérieur à $x$.

**FABS** — **`double fabs (double x)`**

Fournit la valeur absolue de $x$.

**Remarque C99**

La norme C99 introduit d'autres fonctions mathématiques (puissance réelle, logarithme à base 2, fonction `log(1+x)`, fonction « gamma »…) et elle généralise aux types complexes, toutes les fonctions mathématiques à argument flottant.

De plus, elle introduit des fonctions mathématiques « génériques » ; il est alors possible d'utiliser le même nom de fonction, quelle que soit la nature de ses arguments (`float`, `double`, `long double`, `float complex`, `double complex` ou `long double complex`).

## 5 Utilitaires (stdlib.h)

**ATOF** **double atof (const char * chaine)**

Fournit le résultat de la conversion en double du contenu de chaine. Cette fonction ignore les éventuels séparateurs de début et, à l'image de ce que fait le code format %f, utilise les caractères suivants pour fabriquer une valeur numérique. Le premier caractère invalide arrête l'exploration.

**ATOI** **int atoi (const char * chaine)**

Fournit le résultat de la conversion en int du contenu de chaine. Cette fonction ignore les éventuels séparateurs de début et, à l'image de ce que fait le code format %d, utilise les caractères suivants pour fabriquer une valeur numérique. Le premier caractère invalide arrête l'exploration.

**ATOL** **long atol (const char * chaine)**

Fournit le résultat de la conversion en long du contenu de chaine. Cette fonction ignore les éventuels séparateurs de début et, à l'image de ce que fait le code format %ld, utilise les caractères suivants pour fabriquer une valeur numérique. Le premier caractère invalide arrête l'exploration.

**RAND** **int rand (void)**

Fournit un nombre entier aléatoire (en fait pseudo-aléatoire), compris dans l'intervalle [0, RAND_MAX]. La valeur prédéfinie RAND_MAX est au moins égale à 32767.

**SRAND** **void srand (unsigned int graine)**

Modifie la « graine » utilisée par le « générateur de nombres pseudo-aléatoires » de rand. Par défaut, cette graine a la valeur 1.

**CALLOC** **void * calloc (size_t nb_blocs, size_t taille)**

Alloue l'emplacement nécessaire à nb_blocs consécutifs de chacun taille octets, initialise chaque octet à zéro et fournit l'adresse correspondante lorsque l'allocation a réussi ou un pointeur nul dans le cas contraire.

**MALLOC** **void * malloc (size_t taille)**

Alloue un emplacement de taille octets, sans l'initialiser, et fournit l'adresse correspondante lorsque l'allocation a réussi ou un pointeur nul dans le cas contraire.

**REALLOC** **void realloc (void * adr, size_t taille)**

Modifie la taille d'une zone d'adresse `adr` préalablement allouée par `malloc` ou `calloc`. Ici, `taille` représente la nouvelle taille souhaitée, en octets. Cette fonction fournit l'adresse de la nouvelle zone ou un pointeur nul dans le cas où la nouvelle allocation a échoué (dans ce dernier cas, le contenu de la zone reste inchangé). Lorsque la nouvelle taille est supérieure à l'ancienne, le contenu de l'ancienne zone est conservé (il a pu éventuellement être alors recopié). Dans le cas où la nouvelle taille est inférieure à l'ancienne, seul le début de l'ancienne zone (c'est-à-dire `taille` octets) est conservé.

**FREE** **void free (void * adr)**

Libère la mémoire d'adresse `adr`. Ce pointeur doit obligatoirement désigner une zone préalablement allouée par `malloc`, `calloc` ou `realloc`. Si `adr` est nul, cette fonction ne fait rien.

**EXIT** **void exit (int etat)**

Termine l'exécution du programme. Cette fonction ferme les fichiers ouverts en vidant les tampons et rend le contrôle au système, en lui fournissant la valeur `etat`. La manière dont cette valeur est effectivement interprétée dépend de l'implémentation, toutefois la valeur `0` est considérée comme une fin normale.

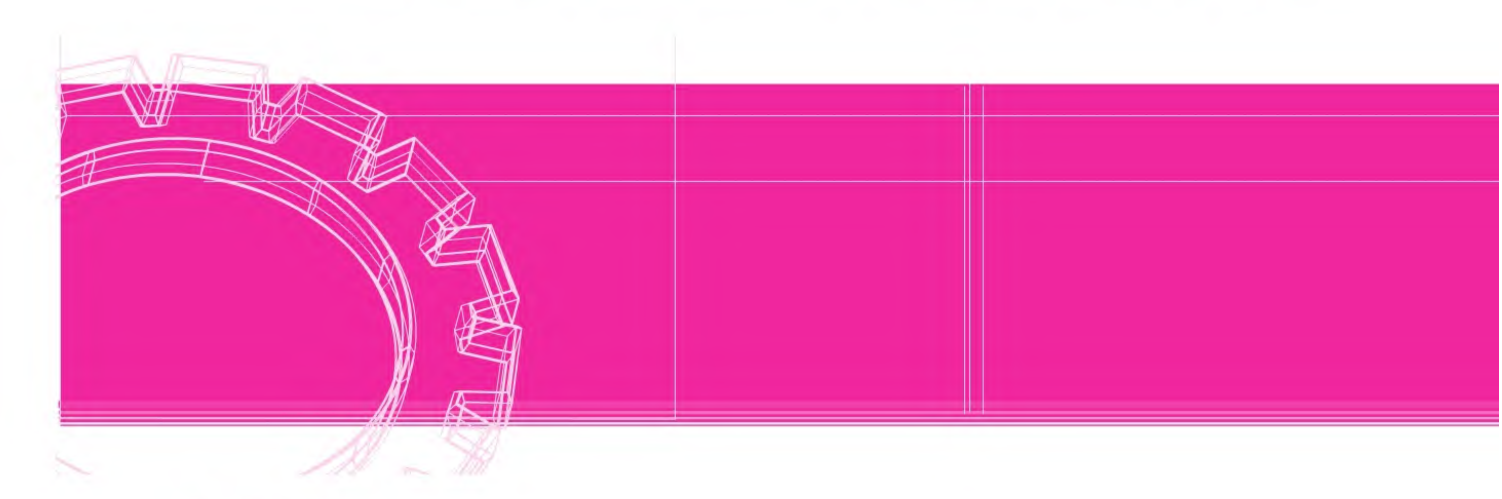
**ABS** **int abs (int n)**

Fournit la valeur absolue de `n`.

**LABS** **long abs (long n)**

Fournit la valeur absolue de `n`.

# Correction des exercices



## CHAPITRE 3

### *Exercice 3.1 :*

a) long 12
b) float 11,75
c) long 4
d) int 0
e) int 1
f) int 1
g) long 5
h) int 1
i) int 0
j) float 1,75
k) float 8,75

### *Exercice 3.2 :*

```
z = a + b ;
```

### Exercice 3.3 :

`n ? (n>0 ? 1 : -1) : 0)` ou `n>0 ? 1 : (n ? -1 : 0)`

### Exercice 3.4 :

```
A : n = 10  p = 10  q = 10  r = 1
B : n = 15  p = 10  q = 5
C : n = 15  p = 11  q = 10
D : n = 16  p = 11  q = 15
```

## CHAPITRE 4

### Exercice 4.1 :

```
A : 543 34.567799
B : 543 34.567799
C : 543 34.567799
D :     34.568  3.457e+01
E : 543
F :     34.56780
```

### Exercice 4.2 :

**a)** n=12, p=45
**b)** n=1234, p=56
**c)** n=1234, p=56
**d)** n=1, p=45
**e)** n=4567, p=89

## CHAPITRE 5

### Exercice 5.1 :

**a)**

```
#include <stdio.h>
main()
{   int i, n, som ;
    som = 0 ;
    i = 0 ;          /* ne pas oublier cette "initialisation" */
```

```
      while (i<4)
         { printf ("donnez un entier ") ;
           scanf ("%d", &n) ;
           som += n ;
           i++ ;                  /* ni cette "incrémentation" */
         }
      printf ("Somme : %d\n", som) ;
   }
```

**b)**

```
   #include <stdio.h>
   main()
   {  int i, n, som ;
      som = 0 ;
      i = 0 ;           /* ne pas oublier cette "initialisation" */
      do
         { printf ("donnez un entier ") ;
           scanf ("%d", &n) ;
           som += n ;
           i++ ;                /* ni cette "incrémentation" */
         }
      while (i<4) ;             /* attention, ici, toujours <4 */
      printf ("Somme : %d\n", som) ;
   }
```

### *Exercice 5.2 :*

```
   #include <stdio.h>
   main()
   {  float note,         /* note courante */
            som,          /* somme des notes */
            moy ;         /* moyenne des notes */
      int num ;           /* numéro note courante */

      som=0 ; num=0 ;
      while ( printf("note %d : ",num+1),
              scanf ("%f", &note), note>=0 )
         {  num++ ;
            som += note ;
         }
```

```
    if (num>0)
       {  moy = som/num ;
          printf ("moyenne de ces %d notes : %5.2f", num, moy) ;
       }
     else printf ("--- aucune note fournie ---") ;
}
```

*Exercice 5.3 :*

```
#include <stdio.h>
main()
{  int nbl ;       /* nombre de lignes */
   int i, j ;
   printf ("combien de lignes : ") ;
   scanf ("%d", &nbl) ;
   for (i=1 ; i<=nbl ; i++)
     { for (j=1 ; j<=i ; j++)
           printf ("*") ;
        printf ("\n") ;
     }
}
```

*Exercice 5.4 :*

```
#include <stdio.h>
#include <math.h>
main()
{
   int n,       /* nombre entier à examiner */
       d ;      /* diviseur courant */
   do
      { printf ("donnez un entier supérieur à 2 : ") ;
        scanf  ("%d", &n) ;
      }
   while (n<=2) ;
   d=2 ;
   while ( (n%d) && (d<=sqrt(n)) ) d++ ;
   if (n%d) printf ("%d est premier", n) ;
       else printf ("%d n'est pas premier", n) ;
}
```

## *Exercice 5.5 :*

```
main()
{
   int u1, u2, u3 ;              /* pour "parcourir" la suite */
   int n ;                       /* rang du terme demandé */
   int i ;                       /* compteur */

   do
     { printf ("rang du terme demandé (au moins 3) ? ") ;
       scanf ("%d", &n) ;
     }
   while (n<3) ;

   u2 = u1 = 1 ;        /* les deux premiers termes */
   i = 2 ;
   while (i++ < n)      /* attention, l'algorithme ne fonctionne */
      { u3 = u1 + u2 ;  /*           que pour n > 2              */
        u1 = u2 ;
        u2 = u3 ;
      }

   printf ("Valeur du terme de rang %d : %d", n, u3) ;
}
```

## *Exercice 5.6*

```
#include <stdio.h>
#define NMAX 10                    /* nombre de valeurs */

main()
{    int i, j ;
          /* affichage ligne en-tête */
     printf ("      I") ;
     for (j=1 ; j<=NMAX ; j++) printf ("%4d", j) ;
     printf ("\n") ;
     printf ("-------") ;
     for (j=1 ; j<=NMAX ; j++) printf ("----") ;
     printf ("\n") ;
```

```
        /* affichage des différentes lignes */
   for (i=1 ; i<=NMAX ; i++)
      { printf ("%4d  I", i) ;
        for (j=1 ; j<=NMAX ; j++)
           printf ("%4d", i*j) ;
        printf ("\n") ;
      }
```

# CHAPITRE 6

## *Exercice 6.1*

```
#include <stdio.h>
void f1 (void)
{  printf ("bonjour\n") ;
}
void f2 (int n)
{  int i ;
   for (i=0 ; i<n ; i++)
       printf ("bonjour\n") ;
}
int f3 (int n)
{  int i ;
   for (i=0 ; i<n ; i++)
       printf ("bonjour\n") ;
   return 0 ;
}
main()
{  void f1 (void) ;
   void f2 (int) ;
   int f3 (int) ;
   f1 () ;
   f2 (3) ;
   f3 (3) ;
}
```

## *Exercice 6.2*

Il affiche : 5 3

## Exercice 6.3

```
#include <stdio.h>
void compte(void)
{
   static long n=0 ;
   static long limit=1 ;
   n++ ;
   if (n>=limit)
      { printf ("** appel %ld fois **\n", limit) ;
        limit *= 10 ;
      }
}
main()
{
   void compte(void) ;
   long nmax=100000 ;
   long i ;
   for (i=1 ; i<=nmax ; i++) compte() ;
}
```

## Exercice 6.4

```
#include <stdio.h>
int acker (int m, int n)
{  if ( (m<0) || (n<0) )
        return(0) ;
   else if (m==0)
        return (n+1) ;
   else if  (n==0)
        return ( acker(m-1,1) ) ;
   else
        return acker ( m-1, acker(m,n-1) ) ;
}
main()
{  int acker (int, int) ;
   int m, n ;
   printf ("donnez m et n : ") ;
   scanf ("%d %d", &m, &n) ;
   printf ("acker ( %d,%d) = %d", m, n, acker(m,n) ) ;
}
```

# CHAPITRE 7

## Exercice 7.1

**a)**

```
#include <stdio.h>
#define NVAL 10                    /* nombre de valeurs du tableau */
main()
{   int i, min, max ;
    int t[NVAL] ;
    printf ("donnez %d valeurs\n", NVAL) ;
    for (i=0 ; i<NVAL ; i++) scanf ("%d", &t[i]) ;
    max = min = t[0] ;
    for (i=1 ; i<NVAL ; i++)
      { if (t[i] > max) max = t[i] ;  /* ou max = t[i]>max ? t[i] : max */
        if (t[i] < min) min = t[i] ;  /* ou min = t[i]<min ? t[i] : min */
      }
    printf ("valeur max : %d\n", max) ;
    printf ("valeur min : %d\n", min) ;
}
```

**b)**

```
#include <stdio.h>
#define NVAL 10                  /* nombre de valeurs du tableau */
main()
{   int i, min, max ;
    int t[NVAL] ;
    printf ("donnez %d valeurs\n", NVAL) ;
    for (i=0 ; i<NVAL ; i++) scanf ("%d", t+i) ;
                                 /* et non  *(t+i) !! */
    max = min = *t ;
    for (i=1 ; i<NVAL ; i++)
      { if (*(t+i) > max) max = *(t+i) ;
        if (*(t+i) < min) min = *(t+i) ;
      }
    printf ("valeur max : %d\n", max) ;
    printf ("valeur min : %d\n", min) ;
}
```

## *Exercice 7.2*

```
void maxmin (int t[], int n, int * admax, int * admin)
{  int i, max, min ;
   max = t[0] ;
   min = t[0] ;
   for (i=1 ; i<n ; i++)
      {  if (t[i] > max) max = t[i] ;
         if (t[i] < min) min = t[i] ;
      }
   *admax = max ;
   *admin = min ;
}
#include <stdio.h>
main()
{  void maxmin (int [], int, int *, int *) ;
   int t[8] = { 2, 5, 7, 2, 9, 3, 9, 4} ;
   int max, min ;
   maxmin (t, 8, &max, &min) ;
   printf ("valeur maxi : %d\n", max) ;
   printf ("valeur mini : %d\n", min) ;
}
```

## *Exercice 7.3*

```
void tri (unsigned char c[] , int nc)
{  int i,j ;
   char ct ;
   for (i=0 ; i<nc-1 ; i++)
      for (j=i+1 ; j<nc ; j++)
         if ( c[i]>c[j] )
            { ct = c[i] ;
              c[i] = c[j] ;
              c[j] = ct ;
            }
}
```

### *Exercice 7.4*

```
void sommat (double * a, double * b, double * c, int n, int p)
{  int i ;
   for (i=0 ; i<n*p ; i++)
     *(c+i) = *(a+i) + *(b+i) ;
}
```

## CHAPITRE 8

### *Exercice 8.1*

```
#include <stdio.h>
#include <string.h>
#define CAR 'e'
#define LGMAX 132


main()
{    char texte[LGMAX+1] ;
     char * adr ;
     int ncar ;
     printf ("donnez un texte terminé par return\n") ;
     gets (texte) ;
     ncar = 0 ;
     adr = texte ;
     while ( adr=strchr(adr,CAR) )
        { ncar++ ;
          adr++ ;
        }
     printf ("votre texte comporte %d fois le caractère %c", ncar, CAR) ;
}
```

### *Exercice 8.2*

```
#include <stdio.h>
#include <string.h>
#define CAR 'e'
#define LGMAX 132
```

```
main()
{   char texte[LGMAX+1] ;
    char * adr ;
    printf ("donnez un texte terminé par return\n") ;
    gets (texte) ;
    adr = texte ;
    while ( adr=strchr(adr,CAR) )  strcpy (adr, adr+1) ;
    printf ("voici votre texte privé des caractères %c\n", CAR) ;
    puts (texte) ;
}
```

### Exercice 8.3

```
#include <stdio.h>
#include <string.h>
#define NBCAR 30
main()
{  char nom[NBCAR+1] ;
   int i ;
   printf ("donnez un nom d'au plus %d caractères : ", NBCAR) ;
   gets(nom) ;
   for ( i=strlen(nom) ; i>=0 ; i--)
      putchar (nom[i]) ;
}
```

### Exercice 8.4

```
#include <stdio.h>
#include <string.h>
#define LGMAX 26
main()
{
  char verbe [LGMAX+1] ;
  char * adfin ;
  char * term[6] = {"e",  "es", "e",  "ons",  "ez",   "ent" } ;
  char * deb[6]  = {"je", "tu", "il", "nous", "vous", "ils"} ;
  int i ;
  do
    { printf ("donnez un verbe du premier groupe : ") ;
      gets (verbe) ;
      adfin = verbe + strlen(verbe) - 2 ;
    }
  while ( strcmp (adfin, "er") ) ;
```

```
     printf ("\nIndicatif présent :\n") ;
     for (i=0 ; i<6 ; i++)
       { strcpy (adfin, term[i]) ;
         printf ("%s %s\n", deb[i], verbe) ;
       }
   }
```

# CHAPITRE 9

## *Exercice 9.1*

```
#include <stdio.h>
#define NPOINTS 5
main()
{  struct point { int num ;
                  float x ;
                  float y ;
                } ;
   struct point courbe[NPOINTS] ;
   int i ;

   for (i=0 ; i<NPOINTS ; i++)
      {  printf ("numéro : ") ; scanf ("%d", &courbe[i].num) ;
         printf ("x      : ") ; scanf ("%f", &courbe[i].x) ;
         printf ("y      : ") ; scanf ("%f", &courbe[i].y) ;
      }

   printf (" **** structure fournie ****\n") ;
   for (i=0 ; i<NPOINTS ; i++)
        printf ("numéro : %d   x : %f   y : %f\n",
                   courbe[i].num, courbe[i].x, courbe[i].y) ;
}
```

## Exercice 9.2

```
#include <stdio.h>
#define NPOINTS 5
struct point { int num ;
               float x ;
               float y ;
             } ;
void lit     (struct point  []) ;  /* ou void lit (struct point *) */
void affiche (struct point  []) ;  /* ou void lit (struct point *) */

main()
{
   struct point courbe[NPOINTS] ;
   lit (courbe) ;
   affiche (courbe) ;
}
void lit (struct point courbe []) /* ou void lit (struct point * courbe) */
{
   int i ;
   for (i=0 ; i<NPOINTS ; i++)
      {  printf ("numéro : ") ; scanf ("%d", &courbe[i].num) ;
         printf ("x      : ") ; scanf ("%f", &courbe[i].x) ;
         printf ("y      : ") ; scanf ("%f", &courbe[i].y) ;
      }
}
void affiche (struct point courbe []) /* ou void affiche
                                         (struct point * courbe) */
{
   int i ;
   printf (" **** structure fournie ****\n") ;
   for (i=0 ; i<NPOINTS ; i++)
   printf ("%d %f %f\n", courbe[i].num, courbe[i].x, courbe[i].y) ;
}
```

# CHAPITRE 10

## Exercice 10.1

```
#include <stdio.h>
#define LGMAX 81
main()
{  char nomfich[21] ;      /* nom de fichier */
   FILE * entree ;
   int num = 1 ;           /* numéro de ligne */
   char ligne [LGMAX] ;    /* tampon d'une ligne */

   printf ("donnez le nom du fichier à lister : ");
   scanf ("%20s", nomfich) ;
   entree = fopen (nomfich, "r") ;
   printf ("  **** liste du fichier %s ****\n", nomfich) ;
   while ( fgets (ligne, LGMAX, entree) )
       {  printf ("%5d ", num++) ;
          printf ("%s", ligne) ;
       }
}
```

**Remarque**

Certaines implémentations demanderont le mode « rt ».

## Exercice 10.2

```
#include <stdio.h>
#define LGNOM     20
#define LGPRENOM 15
#define LGTEL     11
main()
{
   char nomfich[21] ;      /* nom de fichier */
   FILE * sortie ;
   struct { char nom [LGNOM+1] ;
            char prenom [LGPRENOM+1] ;
            int  age ;
            char tel [LGTEL+1] ;
          } bloc ;
```

```
   printf ("donnez le nom du fichier à créer : ");
   gets (nomfich) ;
   sortie = fopen (nomfich, "w") ;
  printf (" --- pour finir la saisie, donnez un nom 'vide' ---\n") ;
  while ( printf ("nom      : "), gets (bloc.nom), strlen(bloc.nom) )
        {  printf ("prénom    : ") ;
           gets (bloc.prenom) ;
           printf ("age       : ") ;
           scanf ("%d", &bloc.age) ; getchar() ;
           printf ("téléphone : ") ;
           gets (bloc.tel) ;
           fwrite (&bloc, sizeof(bloc), 1, sortie) ;
        }
   fclose (sortie) ;
}
```

### Exercice 10.3

```
#include <stdio.h>
#include <string.h>
#define LGNOM    20
#define LGPRENOM 15
#define LGTEL    11
main()
{
   char nomfich[21] ;          /* nom de fichier */
   FILE * entree ;
   struct { char nom [LGNOM+1] ;
            char prenom [LGPRENOM+1] ;
            int  age ;
            char tel [LGTEL+1] ;
          } bloc ;
   char nomcher [LGNOM+1] ;    /* nom recherché */
   int trouve ;                /* indicateur nom trouvé */

   printf ("donnez le nom du fichier à consulter : ");
   gets (nomfich) ;
```

```
    entree = fopen (nomfich, "r") ;
    printf (" quel nom recherchez-vous : ") ;
    gets (nomcher) ;
    trouve = 0 ;

   do
        { fread (&bloc, sizeof(bloc), 1, entree) ;
          if ( strcmp (nomcher, bloc.nom)==0 ) trouve = 1 ;
        }
    while ( (!trouve) && (!feof(entree)) ) ;
    if (trouve)
         { printf ("prénom    : %s\n", bloc.prenom) ;
           printf ("âge       : %d\n", bloc.age) ;
           printf ("téléphone : %s\n", bloc.tel) ;
         }
     else printf ("-- ce nom ne figure pas au fichier --") ;
}
```

### *Exercice 10.4*

```
#include <stdio.h>
#define LGNOM    20
#define LGPRENOM 15
#define LGTEL    11
main()
{
   char nomfich[21] ;          /* nom de fichier */
   FILE * entree ;

   struct { char nom [LGNOM+1] ;
            char prenom [LGPRENOM+1] ;
            int  age ;
            char tel [LGTEL+1] ;
          } bloc ;
   int num ;                   /* numéro de bloc cherché */
   long taille,                /* taille du fichier en octets */
        pos ;                  /* position dans le fichier */

   printf ("donnez le nom du fichier à consulter : ");
   gets (nomfich) ;
```

```
   entree = fopen (nomfich, "r") ;
   fseek (entree, 0, SEEK_END) ; taille = ftell(entree) ;

   printf (" quel numéro recherchez-vous : ") ;
   scanf ("%d",&num) ;

   pos = num * sizeof(bloc) ;
   if ( num<0 || pos >= taille )
         printf ("-- ce numéro ne figure pas dans le fichier ") ;
      else
        { fseek (entree, pos, 0 ) ;
          fread (&bloc, sizeof(bloc), 1, entree) ;
          printf ("nom       : %s\n", bloc.nom) ;
          printf ("prénom    : %s\n", bloc.prenom) ;
          printf ("age       : %d\n", bloc.age) ;
          printf ("téléphone : %s\n", bloc.tel) ;
        }
}
```

# CHAPITRE 11

## *Exercice 11.1*

```
#include <stdio.h>
#include <stdlib.h>
typedef struct element { int num ;
                         float x ;
                         float y ;
                         struct element * suivant ;
                       } s_point ;
void creation (s_point * * adeb) ;
void liste    (s_point *  debut) ;
main()
{
   s_point *  debut ;
   creation (&debut) ;
   liste(debut) ;
}
```

```
void creation (s_point * * adeb)
{
   int num ;
   float x, y ;
   s_point * courant ;
   * adeb = NULL ;
   while ( printf("numéro x y : "),
           scanf ("%d %f %f", &num, &x, &y), num)
       { courant = (s_point *) malloc (sizeof(s_point)) ;
         courant -> num     = num ;
         courant -> x       = x ;
         courant -> y       = y ;
         courant -> suivant = * adeb ;
         * adeb = courant ;
       }
}

void liste (s_point * debut)
{
   printf ("  **** liste de la structure ****\n") ;
   while (debut)
     { printf ("%d %f %f\n", (debut)->num, (debut)->x, (debut)->y) ;
        debut = (debut)->suivant ;
      }
}
```

# Index

## Symbols



## A

# B



# C

## D



## E



## F

## G



## I

## L



## M



## N



## O



## P

## R



## S

## T



## U



## V



## W